Scalable Computation of Streamlines on Very Large Datasets *

Dave Pugmire
Oak Ridge National Laboratory
PO Box 2008 MS6016
Oak Ridge, TN 37831-6016
pugmire@ornl.gov

Hank Childs
Lawrence Berkeley National Laboratory
One Cyclotron Road
Berkeley, CA, 94701
hchilds@lbl.gov

Christoph Garth
University of California, Davis
One Shields Ave.
Davis, CA, 95616
cgarth@ucdavis.edu

Sean Ahern
Oak Ridge National Laboratory
PO Box 2008 MS6016
Oak Ridge, TN 37831-6016
ahern@ornl.gov

Gunther H. Weber
Lawrence Berkeley National Laboratory
One Cyclotron Road
Berkeley, CA, 94701
ghweber@lbl.gov

ABSTRACT

Understanding vector fields resulting from large scientific simulations is an important and often difficult task. Streamlines, curves that are tangential to a vector field at each point, are a powerful visualization method in this context. Application of streamline-based visualization to very large vector field data represents a significant challenge due to the non-local and data-dependent nature of streamline computation, and requires careful balancing of computational demands placed on I/O, memory, communication, and processors. In this paper we review two parallelization approaches based on established parallelization paradigms (static decomposition and on-demand loading) and present a novel hybrid algorithm for computing streamlines. Our algorithm is aimed at good scalability and performance across the widely varying computational characteristics of streamline-based problems. We perform performance and scalability studies of all three algorithms on a number of prototypical application problems and demonstrate that our hybrid scheme is able to perform well in different settings.

Keywords

visualization, streamlines, flow, scaling, parallel

*(Does NOT produce the permission block, copyright information nor page numbering). For use with ACM_PROC_-ARTICLE-SP.CLS. Supported by ACM.

1. INTRODUCTION

Simulations on the current generation of supercomputers are producing data sets of unprecedented scale. To achieve the fundamental goal of scientific insight from the resulting very large datasets, a variety of problems must be addressed pertaining to their storage and handling. For simulations that involve vector fields, integral curves, or streamlines, are one of the most illuminating techniques to obtain insight; they are a cornerstone of visualization and analysis across a great 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 application-domain vector fields, such as transport and mixing in fluid flows. However, calculating integral curves in large data presents a significant challenge because their calculation is non-local and data dependent. Thus, leveraging parallel computational resources to achieve scalable and well-performing visualization in this setting requires optimal parallelization strategies that adapt smartly to the widely varying characteristics of integral curve computation.

Starting from these observations, in this work, we explore a number of such algorithms and introduce a novel adaptive parallelization scheme that addresses some of the unique problems of parallel streamline computation and provides improved performance. While efforts towards parallel streamlines are typically focused mostly on the size of the considered vector field data or are aimed at accelerating a pre-chosen visualization technique, the discussion and parallelization scheme we present here are aimed at a very general setting. Beyond taking into account data size, we also focus on an analysis of the different characteristics of integration-based problems, such as for example the number of integral curves to be computed, to guarantee good parallelization and scalability over a wide range of situations encountered in applications.

Overall, we provide a comprehensive analysis of three different parallel strategies. The first two studied schemes, static data decomposition and an on-demand loading of required data, refer to established parallelization paradigms. The novel parallelization strategy we present here is a hybrid method, aimed at optimal load balancing with minimal I/O and good scalability. The key design aspect of our new method is to exploit coherency in streamline computation to allow an optimal workload distribution across processors. To obtain insight into the respective characteristics and performance of all three approaches, we study their behavior on a number of representative application problems involving integral curve computation. Besides documenting the performance benefit available from our hybrid scheme, we aim at providing data points and heuristics that can generally help scientists choose a parallelization strategy depending on the parameters of an integration-based problem.

The remainder of the paper is structured as follows. After briefly examining the visualization background of streamline-based methods and reviewing previous work in Section 2, we discuss the general setting that we base our analysis on in Section 3. This includes a detailed description of the considered application problems and the different degrees of freedom that determine their respective computational characteristics. We then describe the parallelization strategies we consider here and introduce a new hybrid scheme in Section 4, together with a consideration of some of the theoretical performance aspects of all three methods. Section 5 reports performance measurements and provides a discussion of the observed behavior and scalability of the different strategies. Finally, in Section 6, we provide a number of heuristic decision guidelines based on the observed measurements that can be used to pick one of the presented parallelization strategies for future problems.

While the discussion in this paper is largely implementation-agnostic, we wish to point out that the methods described here are easily implemented using commonly used visualization pipeline architectures. Specifically, we provide a full implementation of the present work in the open-source VisIt [1] visualization system that is freely available to the research community.

2. BACKGROUND AND PREVIOUS WORK

2.1 Streamlines

For a stationary vector field v that does not depend on time, an integral curve is called a *streamline* and is given by the ordinary differential equation

$$\dot{S}(t) \;=\; v(S(t,x)) \qquad \text{and} \qquad S(t_0) \;= x_0. \qquad (1)$$

Hence it describes a parameterized curve that starts at the *seed point* x_0 and is tangent to v over its parameter interval $[t_0, t_1]$ for $t_0 < t_1$.

In the discrete setting we are concerned with in this work, streamlines are approximated using numerical integration methods to approximate the describing ordinary differential equations. There is an extensive body of work on this topic, and we refer the interested reader to [11]. In our streamline implementation, we use an integration scheme of Runge-Kutta type with adaptive stepsize control as proposed by Dormand and Prince [18]. However, the material discussed in the remainder of this paper equally applies to most other customarily used integration schemes.

The visualization and analysis of vector fields is an active research area, and so-called *integration-based* techniques that derive vector field visualization from integral curves have progressed well beyond the direct depiction of individual streamlines or a small subset of them [16]. Integral surface techniques [10, 13] compute and visualize a surface consisting of all streamlines emanating from a common curve, while flow volumes examine the behavior of entire volumes of particles [23, 2]. Topological methods, on the other hand, aim at extracting the structural skeleton of a vector field by considering the dynamical system induced by it and computing critical points and stable and unstable manifolds. More recently, the notions of *Finite-Time Lyapunov Exponents* and *Lagrangian Coherent Structures* [12, 15, 19] were introduced to allow for an accurate structural analysis of time-varying vector fields. These *Lagrangian* methods, which can require many thousands to millions of streamlines, are built on observing the separation between closely neighboring particles as they are advected, and coherent structures are then identified by lines and surfaces along which this separation is maximal.

2.2 Parallel Streamline Computation

The parallel solution of streamline-based problems has been considered in previous work using different approaches. An early treatment of the topic was given by Sujudi and Haimes [21], who made use of distributed computation by splitting a block in consideration into several sub-blocks and assigning each processor one such sub-block. A streamline is communicated among processors as it traverses different sub-blocks.

Other examples of applying parallel computation to streamline-based visualization include the use of multiprocessor workstations to parallelize integral curve computation (e.g. [14]), and research efforts were focused on accelerating specific visualization techniques [4]. Similarly, PC cluster system were leveraged in to accelerate visualization of time-varying Line Integral Convolution volumes [17] or particle visualization for very large data [7].

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, [22] presented a technique to compute streamlines in large unstructured grids using an octtree partitioning of the vector field data for fast fetching during streamline construction small memory footprint. Taking a different approach, Bruckschen et al. [3] describe a technique for real-time particle traces of large time-varying data sets, by isolating all integral curve computation in a pre-processing 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. [24] 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, loss of the ability to choose arbitrary, user-defined seed-points, which is often necessary when using streamlines for data analysis as opposed to getting a qualitative data overview. Chen and Fujishiro [6] apply a spectral decomposition using a vector-field derived anisotropic differential operator to achieve a similar goal.

Two of the methods discussed in this paper (static partitioning and load on demand) are roughly similar to previous work in that they make use of straightforward parallelization strategies to achieve parallel streamline computation. In contrast, our novel hybrid parallelization scheme parallelizes over both streamlines and data size simultaneously, and is thus able to provide improved performance over a wide range of typical scenarios for streamline-based visualization (cf. Sections 3.1). Consequently, the material we provide in the following is intentionally general and does not imply a specific visualization choice or algorithm where advantage could be taken of specific data partitions or seed sets.

Furthermore, we are explicitly aiming at treating unmodified and pre-partitioned data, as output from a simulation. Since a global pre-analysis and re-partitioning of the considered vector field data is computationally expensive and often infeasible due to storage requirements, we do not consider methods requiring such analysis here. Rather, we wish to enable application scientists to apply integration-based visualization to their simulation or measurement data in an "out-of-the-box" fashion. Finally, we note that the algorithms we describe are ideal for integration with the current generation of distributed memory parallel visualization tools, where much of the previous work, for example the work described in [24], would require such tools to undergo large architectural changes.

3. SETTING

In this paper, we are aiming to both qualify and quantify the performance of three different parallelization strategies for streamline computation. Before we provide a discussion of the latter in Section 4, we will first discuss the strongly varying characteristics of integration-based application problems that are likely to have a significant impact on the performance of individual schemes and introduce a number of representative application problems that illustrate them.

3.1 Problem Classification

We classify streamline-based problems according to the four criteria that are described in the following:

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. This scenario typically corresponds to a problem where a large number of streamlines must be computed over a relatively small dataset, such as is often encountered when applying Lagrangian analysis or statistical analysis of integral curves or particle trajectories. Conversely, *large* data cannot be loaded in its entirety and thus requires a form of on-demand loading of required parts of the data. Here, adaptive distribution of data over the available parallel resources and optimal scheduling and dispatch of integral curve computation are necessary traits of a well-performing parallelization approach.

Seed Set Size. If the problem at hand requires only the computation of a few tens to a hundred streamline, 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.

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 domain. This results in integral curves traversing the entire data set. Hence, the seed set distribution determines strongly if performance stands to gain most from parallel computation, data distribution, or both.

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 towards 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. This dependency of streamline computation on the underlying vector field is both counterintuitive and hard to identify without conducting prior analysis to determine the field structure as is done e.g. in [24]. While such analysis can be useful for specific problems, in the context of this paper we are considering a more general setting and do not wish to burden a user with the selection of an appropriate scheme and the extensive computation and storage requirements it entails.

Overall, these problem characteristics determine to what extent a given integration-based problem can profit from parallel computation and data distribution. We next describe a

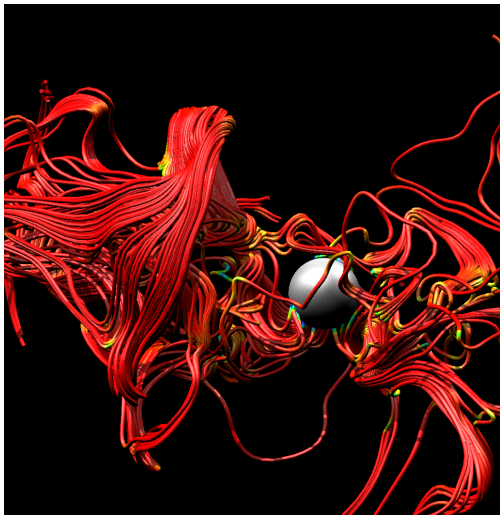

Figure 1: Streamlines in the astrophysics dataset seeded outside the proto-neutron star illustrate the nature of the complex magnetic field inside the supernova shock front.

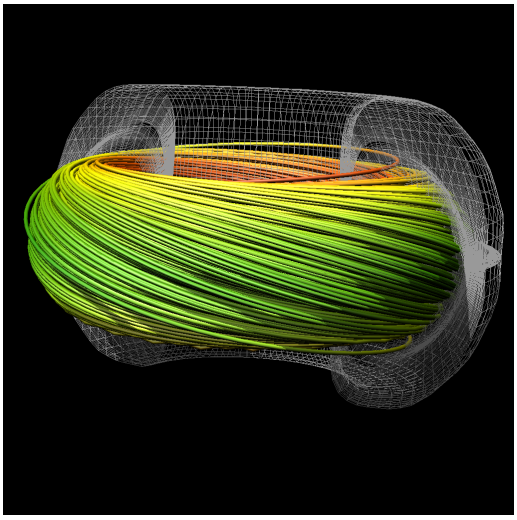

Figure 2: Streamlines show the flow of the magnetic field inside the toroidal plasma chamber.

number of prototypical real-world application problems that exhibit varying characteristics and upon which we base our performance studies.

3.2 Application Problems

To test the effectiveness of the three streamline algorithms outlined below, we apply them to a number of problems that represent typical application scenarios. The datasets include astrophysics, magnetically confined fusion, and thermal hydraulics simulations on regular grids[1]; for each dataset, we consider initial conditions that are both sparse and dense in the respective vector field domains. All analysis test runs for this paper were performed on JaguarPF, a 149,296 processor Cray XT5, located at Oak Ridge National Laboratory.

Astrophysics / Supernova Simulation. For the astrophysics case study, we create streamlines in the magnetic field surrounding a solar core collapse resulting in a supernova. See Figure 1. The vector field used is derived from the magnetic field computed by a GenASiS simulation [8], a multi-physics code being developed for the simulation of astrophysical systems involving nuclear matter [5]. GenASiS computes the magnetic field at each cell face. For the purposes of this scaling 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. We then resample these vectors onto a grid of various sizes to measure performance of the streamline generation algorithms. For the purpose of this scaling study, we sampled the magnetic field onto 512 blocks with 1 million cells per block. Streamlines performance is studied for both sparse and dense seed points sets.

Tokamak / Magnetically Confined Fusion. The second dataset we consider here results from a simulation of magnetically confined fusion in a tokamak device. The simulation was performed using the NIMROD code [20]. This dataset has the unusual property that most streamlines are approximately closed and traverse the torus-shaped vector field domain repeatedly (see Figure 2). However, there are also streamlines that exhibit chaotic behavior and traverse the entire domain as integration time approaches infinity. This is of interest to the material presented here as highly localized streamlines can diverge strongly over time.

For the purposes of this scaling study, the original mesh is resampled onto 512 blocks with 1 million cells per block, and again, streamline performance is studied for both sparse and densely seeded point sets.

Thermal Hydraulics. Figure 3 illustrates streamlines in 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. Non-optimal mixing can be caused by long-lived recirculation zones that effectively isolate certain regions of the domain from heat exchange.

The time-varying simulation was performed using the Nek5000 code[9] on a regular grid of twenty-three million elements. Streamlines are seeded according to two application scenarios. First, streamlines are placed uniformly through the volume to show areas of high velocity, areas of stagnation, and areas of recirculation. Second, we seed the streamlines densely around one of the inlets, to see the behavior of particles entering through the inlet. The resulting streamlines illustrate the turbulence in the immediate vicinity of the inlet. Figure 4 depicts this using a stream surface, which are well suited to this type of visualization. To replicate the conditions of stream surface computation for scalability analysis, we have seeded 22,000 streamlines in the shape of a circle immediately around the inlet.

[1]While results for regular grids are presented in this work, the algorithms discussed also work on arbitrary grids.

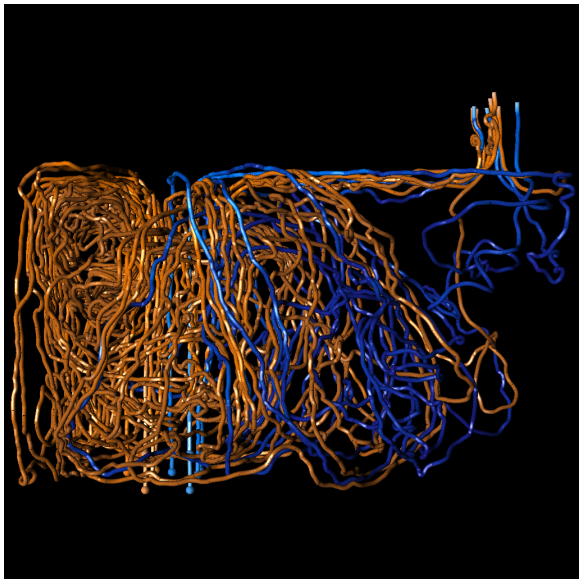

Figure 3: Streamlines in the thermal hydraulics dataset illustrate illustrate how the water from twin inlets mix in a box. One inlet introduces warm water (orange), while the other introduces cold water (blue). Although the warm water dominates one side of the box, both temperatures mix before ultimately exiting through the outlet in the upper right.

4. PARALLELIZATION STRATEGIES

In all algorithms, the problem mesh is decomposed into a number of spatially disjoint blocks. Each block may or may not have ghost cells for connectivity purposes. Each block has a time step associated with it, thus two blocks that occupy the same space at different times are considered independent. The three primary algorithms 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 presented in Section 3.

4.1 Static Allocation

This algorithm is a parallelization across the components of the mesh (blocks). In this algorithm we statically allocate blocks to processors such that the first of n processors is assigned the first $1/n$ of the blocks, the next processor the second $1/n$ of the blocks, etc. Each streamline is integrated until it leaves the blocks owned by the processor. As each streamline moves between blocks, it is communicated to the processor that owns the block in which it currently resides. A globally communicated streamline count is maintained so that all processors may monitor how many streamlines have yet to terminate. Once the count goes to zero, all processors terminate.

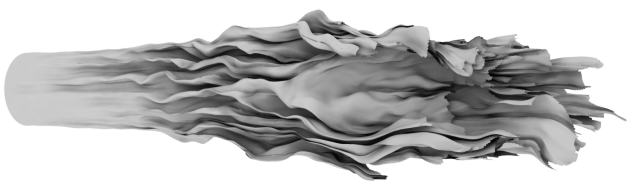

Figure 4: A stream surface in the thermal hydraulics dataset illustrates the strong turbulence in flow leaving an inlet.

4.2 Load On Demand

This algorithm is a parallelization across the streamlines. In this algorithm, we split up the initial seed points evenly among the processors, with $1/n$ of the streamlines assigned to each of the n processors, grouped by block to enhance data locality. Each processor integrates the streamlines assigned to it until streamline termination. As streamlines move between blocks, each processor loads the appropriate block into memory into an LRU (least-recently used) cache. In order to minimize I/O, each processor integrates all streamlines to the edge of the loaded blocks, loading a block from disk only when there is no more work to be done on the in-memory blocks. Each streamline is only ever owned by one processor, though blocks may be loaded by multiple processors. Each processor terminates independently when all of its streamlines have terminated.

The Load On Demand algorithm makes use of caching of blocks in a LRU fashion; old blocks are discarded if available main memory is insufficient to accommodate new blocks that are required to continue streamline integration. Clearly, having more main memory available decreases the need for I/O operations.

4.3 Hybrid Master/Slave

This algorithm is a hybrid between Static Allocation and Load On Demand. In this algorithm, we dynamically assign both streamlines and blocks to processors in an attempt to load balance on the fly based upon the processor workloads and the nature of the vector field. It attempts to keep all processors busy while also minimizing I/O by choosing either to communicate streamlines to other processors or to have processors load duplicate blocks based on heuristics.

Since detailed knowledge of flow is often unpredictable or unknown, the Hybrid Master/Slave algorithm was designed to adapt during the computation to concentrate resources where they are needed, distributing streamlines where needed, and/or duplicating blocks when needed. Through this, we achieve parallelization across data size and streamlines simultaneously, and are able to adapt to the strongly varying characteristics of integration-based problems.

In this algorithm we have a master process that coordinates the workloads of a fixed set of slave processors. The master makes initial assignments to the slaves based on the initial seed point assignment. As work progresses, the master monitors the length of each slave's work queue and the blocks that are loaded and reassigns streamline computation to balance both slave overload and slave starvation. When the master determines that all streamlines have terminated, it instructs all slaves to terminate.

Algorithm 1 Slave process

```
PROCEDURE Slave
  while (not done) do
    while (streamlines to integrate) do
      if Last streamline then
        state ← Calculate slave state
        Master ← state
      end if
      Compute a streamline
    end while
    Process messages from Master
  end while
END Procedure
```

For scalable performance, we introduce the concept of multiple masters, allowing for multiple groups of slaves doing work on different portions of the problem. The multiple masters coordinate balancing the work between them, and each master handles a number of slaves W. We typically use one master per $W = 32$ slaves. Depending on the machine and network characteristics, a different number might be chosen here.

The Hybrid Master/Slave algorithm is clearly the most complex of the algorithms. The design of the slave process is quite simple. Each slave continuously advances streamlines that reside in blocks that are loaded. Similarly to Static Allocation, blocks are cached to the extent permitted by main memory. When the slave can advance no more streamlines or is out of work, it sends a status message to the master and waits for further instruction. In order to hide latency, the slave sends the status message before it advances the last streamline. At each iteration, the slave checks for incoming instructions and streamlines. Initially, each slave is assigned $N = 10$ streamlines. Pseudocode for the slave process is shown in Algorithm 1.

The master is responsible for maintaining information on each slave and managing their workloads. At regular intervals, each slave sends a status message to the master so that it may keep accurate global information. This status message includes the set of streamlines owned by each slave, which blocks those streamlines currently intersect, which blocks are currently loaded into memory on that slave, and how many streamlines are currently being integrated. Load balancing is achieved by observing a slave overload limit N_O, and not reassigning streamlines if the total workload would rise above this limit, which we typically choose as $N_O = 20 \times N$ to obtain good results. Similarly, streamlines are not migrated from a slave that has a significant number N_L of outstanding streamlines in the same block; rather, it is faster for the slave to load this block and perform integration in it itself. In our experiments, we have obtained good results with $N_L = 40$. Together, N_O and N_L ensure a balance between computation, I/O and communication, and they allow tuning the algorithm towards specific hardware characteristics such as slow network communication between the slaves.

An incoming status message from a slave indicates that the slave cannot perform more work. The master then makes new assignments based on the following 5 basic rules:

Assign$_{\text{loaded}}$: Master sends N seed points in block B to a slave that has block B loaded.

Assign$_{\text{unloaded}}$: Master sends N seed points in block B to a slave. Slave loads block B.

Send$_{\text{force}}$: Master instructs slave S_1 to send streamlines in block B to slave S_2. The master only uses this rule if its use will not increase the load on S_2 above N_O.

Send$_{\text{hint}}$: Master sends a hint to slave S_1 to offload streamlines from a given set of blocks to slave S_2 when appropriate. If S_1 does not have any appropriate streamlines to send, it ignores the hint. This rule allows the slaves some measure of autonomy for efficiency purposes.

Load: Master instructs a slave to load block B.

Note that all slaves receive their initial allocation of work through the Assign$_{\text{unloaded}}$ rule.

At initialization, the streamline seedpoints are distributed equally to each master process. The master algorithm maintains a set of slave records, one record for each slave process. When a slave no longer has work to perform the state is communicated to the master where work decisions are made. Each time the state of the slave is updated, the rules described above are applied. The rules are applied in order, as described below.

When slave status updates arrive, the master identifies the set of slaves with no work to do, if any, S_W. For each slave S in S_W try to assign work by following this sequence in order, terminating the sequence when slave S has been assigned new work:

1. Through the use of the Send$_{\text{force}}$ rule, instruct S to offload streamlines in unloaded blocks to other slaves that have the block loaded.
2. If S has more than N_L streamlines in any unloaded block, instruct S to load the block using the Load rule.
3. The application of the Load rule above modifies the set of blocks loaded by the group of slaves. Therefore, check to see if other slaves can send streamlines in unloaded blocks using the Send$_{\text{force}}$ rule.
4. If the master has any seed points in a block loaded by S, send N seed points from that block using the Assign$_{\text{loaded}}$ rule.
5. If the master has any seed points, send N seed points from any block using the Assign$_{\text{unloaded}}$ rule. S loads the block.
6. If S still has no work assigned, instruct S to load the block populated with the most streamlines using the Load rule.

7. Identify the slaves with the most streamlines to process, S_B. Randomly select one and send a message using the $\text{Send}_{\text{hint}}$ rule that S can be supplied with work.

After any rule has supplied a slave with new work, the slave is removed from S_W and not considered for additional work assignemnts until the slave has completed the newly assigned work and sends a new update status back to the master. As each rule is applied, the master updates its status records for each affected slave. Based on the new work assignments, the master knows the individual and collective state of the group of slaves and is able to make decisions as new status updates arrive. It is possible that after these steps are taken, some slaves may still not have work to do. However, the next time another slave posts a status, the collective state of the work group changes and there is another opportunity for work assignment.

5. PERFORMANCE DISCUSSION

To compare the efficiency of our parallelization strategies, we apply each of the three algorithms to several representative datasets and seed point distribution scenarios and measure various aspects of performance. Because each algorithm parallelizes differently over streamlines and blocks, it is insightful to analyze the performance not only of total running time but also other key metrics that are impacted directly by the parallelization strategy choices.

As an overall metric, we consider total run time or wall clock time of the algorithm. This metric includes the total time to solution of each algorithm, including streamline computation, I/O and communication. Although of most interest to the end user, this metric alone is not sufficiently fine-grained to give insight into the performance of an algorithm on a given dataset with a given initial seed set. To give more fine-grained insight into performance, we also analyze communication, I/O, and block management.

Communication is a difficult metric to report and analyze since all communication in our algorithms is asynchronous. However, to measure the impact of parallelization that involves communication, we measure the time required to post *send* and *receive* operations and associated communication management.

To measure the impact of I/O upon parallelization, time spent reading blocks from disk is recorded as well as the number of times blocks are loaded and purged. Because not all the blocks will fit into memory, a LRU cache, with a user defined upper bound, is implemented to handle block purging. To measure the efficiency of this aspect of the algorithm, we define E, *block efficiency*, as the ratio of the number of blocks loaded minus the number of blocks purged to the number of blocks loaded.

$$E = \frac{B_L - B_P}{B_L} \tag{2}$$

For each algorithm, we run using various processor counts on several representative datasets described in Section 3.2

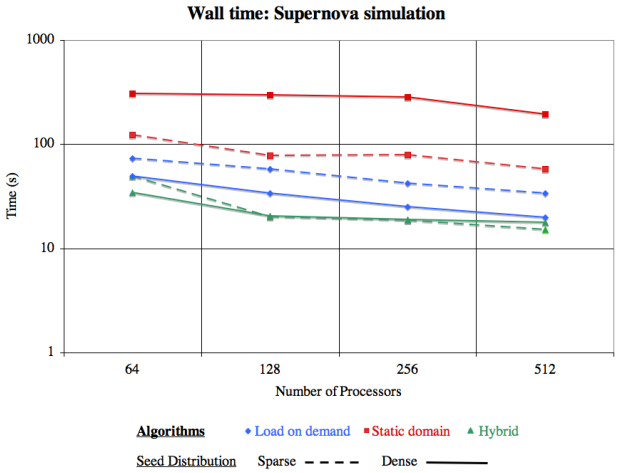

Figure 5: A logarithmic scale plot of wall clock times for all three algorithms, as applied to the astrophysics dataset with both sparse and dense initial seed points.

and several initial seeding conditions. To compare different parallelization strategies, we compare performance on fixed problems and processor counts while varying the streamline algorithm.

5.1 Astrophysics simulation

For the astrophysics case study, we create streamlines from 20,000 seed points with both sparse and dense initial seed point placement.

For the astrophysics dataset, the graph in Figure 5 clearly demonstrates the relative time performance of the three algorithms, with the Hybrid Master/Slave algorithm demonstrating better performance than either Load On Demand or Static Allocation for both spatially sparse and dense initial seed points. However, even at 512 processors, the difference between Hybrid Master/Slave and Static Allocation for the spatially sparse seed point set is only a factor of 3.8, so we must look at other metrics to determine a winner. An examination of total time spent in I/O, as shown in Figure 6, is particularly instructive. In this graph, we see that the Hybrid Master/Slave algorithm performs very close to the ideal, as exemplified by the Static Allocation algorithm. Though Load On Demand performs closely to Hybrid Master/Slave from a time point of view, it spends an order of magnitude more time in I/O for both seed point initial conditions.

We next consider block efficiency (see Equation 2), as shown in Figure 7 for all three algorithms. Static Allocation performs ideally, loading each block once and never purging. Load On Demand performs least efficiently as blocks are loaded and reloaded many times. The performance of the Hybrid Master/Slave algorithm is close to ideal for both sparse and dense seed points. The ability of the Hybrid Master/Slave algorithm to perform so well for both seed point sets is very encouraging.

It is also useful to consider the time spent in communication.

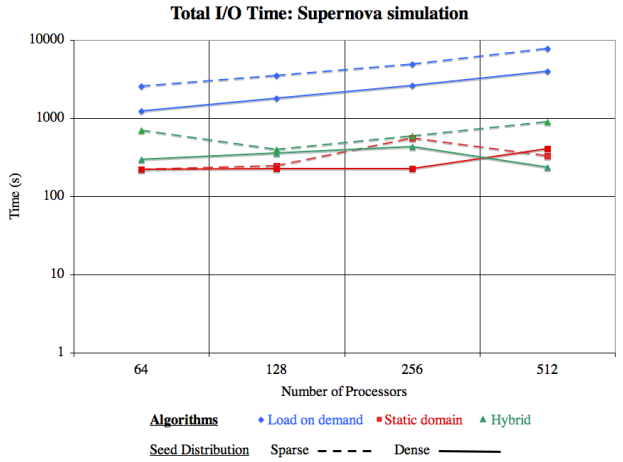

Figure 6: A logarithmic scale plot of total time spent doing I/O for all three algorithms, as applied to the astrophysics dataset with both sparse and dense initial seed points.

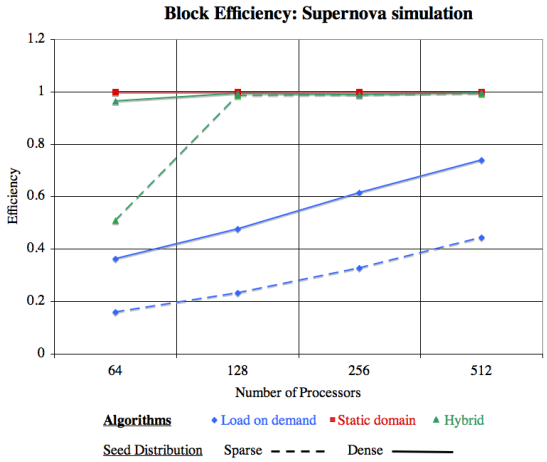

Figure 7: Block efficiency for all three algorithms, as applied to the astrophysics dataset with both sparse and dense initial seed points.

Figure 8 shows the communication pattern for both initial seed point conditions for Hybrid Master/Slave vs. Static Allocation. For the sparse initial condition, Static Allocation performs approximately 20 times more communication than Hybrid Master/Slave as streamlines are sent between the processors that own the blocks. This trend remains even as the number of processors is scaled up. For the dense initial condition, the separation increases by another order of magnitude as Static Allocation performs between 165 and 340 times more communication as the processor count increases. This is because the ratio of blocks needed to total blocks decreases and large numbers of streamlines must be communicated to the processors that own the blocks. (Obviously, no communication occurs with the Load On Demand algorithm.)

5.2 Magnetically confined fusion simulation

For the magnetically confined fusion case study, we create streamlines from 10,000 seed points with both sparse and dense initial seed point placement.

For the fusion dataset, the graph in Figure 9 demonstrates the relative time performance of the three algorithms. In the fusion dataset, the nature of the vector field leads to some interesting performance results. The magnetic field in a fusion simulation rotates within the toroidal containment core. Because of this, regardless of seed placement, the streamlines tend to fill the interior of the torus fairly uniformly. Static Allocation and Hybrid Master/Slave perform nearly identically for both initial conditions. Load On Demand performs poorly for spatially sparse seed points, but very competitively with Static Allocation and Hybrid Master/Slave for a dense seed point set. In the case of Load On Demand with a dense seed point set, good performance is obtained because the working set of active blocks fits inside memory and few blocks must be purged to advance the streamlines around the toroidal core. An analysis of wall clock time does not clearly indicate a dominant algorithm between Static Allocation and Hybrid Master/Slave, so further analysis is warranted. But it is encouraging that Hybrid Master/Slave can adapt to both initial conditions.

The graph of total I/O time is shown in Figure 10. In both seed point initial conditions, as expected, Load On Demand performs more I/O. However, Load On Demand does not have the communication costs and latency of the other two algorithms and so for the case of the dense seed point initial condition, is able to overcome the I/O penalty to show good overall performance.

The graph of total communication time is shown in Figure 11. For a dense seed point set, communication is very high for the Static Allocation algorithm. Since the streamlines tend to be concentrated in an isolated region of the torus, many streamlines must be communicated to the block owning processors. For a sparse seed set, the streamlines are more uniformly distributed and communication costs are therefore lower.

The graph of block efficiency is shown in Figure 12. It is interesting to note that the block efficiency of Hybrid Master/Slave is less than in the astrophysics case study. However, overall performance is still very strong. This indicates that for this particular dataset, better overall performance dictates that more blocks should be replicated across the set of resources.

5.3 Thermal hydraulics simulation

For the the sparse case in the thermal hydraulics simulation, we distributed 4,096 seed points evenly on a 16x16x16 grid throughout the box. For the densely distributed case we placed 22,000 seed points around one of the water inlets in the simulation. The graph for total wall clock time is shown in Figure 13. In the sparse case, all three algorithms have similar run-time, consistent with the performance in the previous sections. Note that the performance of all three algorithms is remarkably similar (within a few seconds of

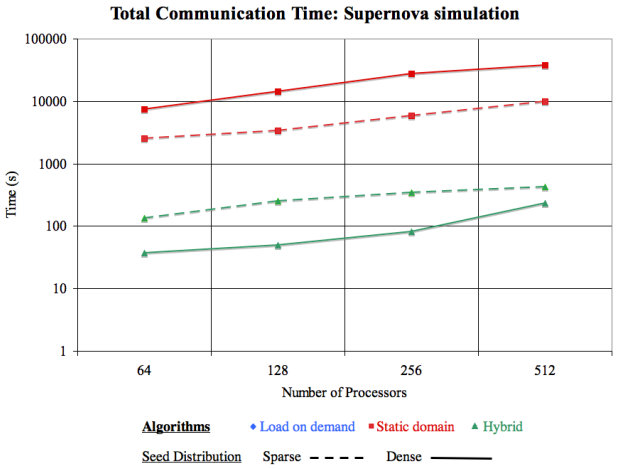

Figure 8: A logarithmic scale plot of total time spent performing communication for all three algorithms, as applied to the astrophysics dataset with both sparse and dense initial seed points.

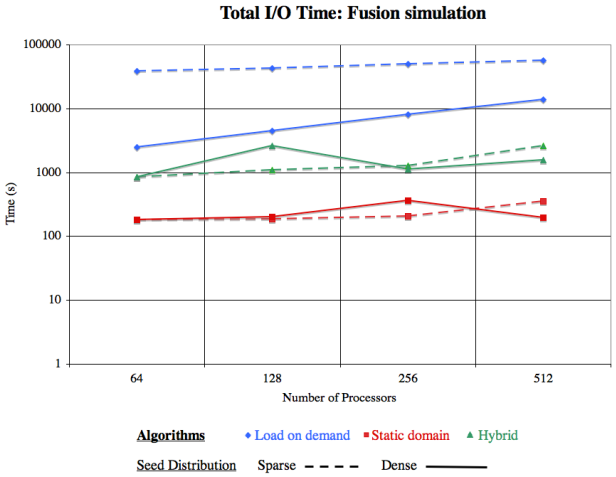

Figure 10: A logarithmic scale plot of total time spent doing I/O for all three algorithms, as applied to the fusion dataset with both sparse and densely placed seed points.

each other) with 512 processors, because this use case is not overly taxing.

The dense seed point case, however, is much more interesting. First, the Static Allocation algorithm ran out of memory and was unable to run. This is because all 22,000 seed points were being processed on a single processor. Even if sufficient memory were available, we would still see tremendous load imbalance, because one processor would be advecting streamlines while the others spin idle. This example illustrates how poorly suited this algorithm is for dense seed point configurations. Also, recall that in Section 3, we identified that such configurations lead to interesting analysis and pictures, so this fundamental limitation of the algorithm

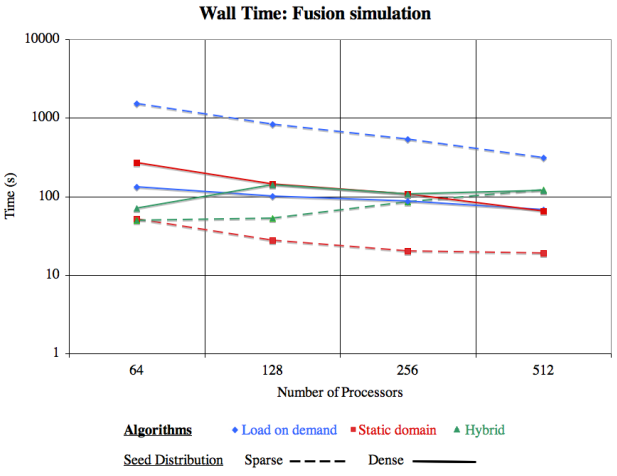

Figure 9: A logarithmic scale plot of wall clock time for all three algorithms, as applied to the fusion dataset with both sparse and dense initial seed points.

is noteworthy.

Further, note that the Load On Demand algorithm outperformed the Hybrid Master/Slave algorithm in the dense case. To understand this, it is important to understand the balance between computation and I/O in this problem. All of the seed points are located in the same portion of the data set, meaning that very little data needs to be read off disk. Because there are so many seed points, the large majority of the execution time is spent doing particle advection. Also, because we only integrated the streamlines a short distance, the streamlines did not travel very far and hence it was not necessary to read many new domains in off disk. This explains the excellent scaling of Load On Demand, because, although data is read redundantly (Load On Demand's major flaw), not much data needs to be read in overall. (Note that Load On Demand's I/O times, Figure 14, are not scaling, but that its wall time, Figure 13, does appear to scale, meaning that I/O costs are minor in total execution time.) Essentially, because there are so many streamlines, the I/O time is hidden altogther. So, in this boundary case, Hybrid Master/Slave performs worse, because it doesn't have all of its processors working at full efficiency, as Load On Demand does.

6. GENERAL ALGORITHM CHARACTERISTICS

As presented in Section 4, the three algorithms we present parallelize across different axes. Load On Demand parallelizes across streamlines, loading blocks as needed with no communication and the associated latency. This algorithm is well suited to datasets that can fit largely in memory or that exhibit flow that is free of vortex-type features larger than the block size. The disadvantage of this algorithm is that it can become I/O bound.

Static Allocation parallelizes across blocks, communicating streamlines as needed, and as such, performs minimal I/O.

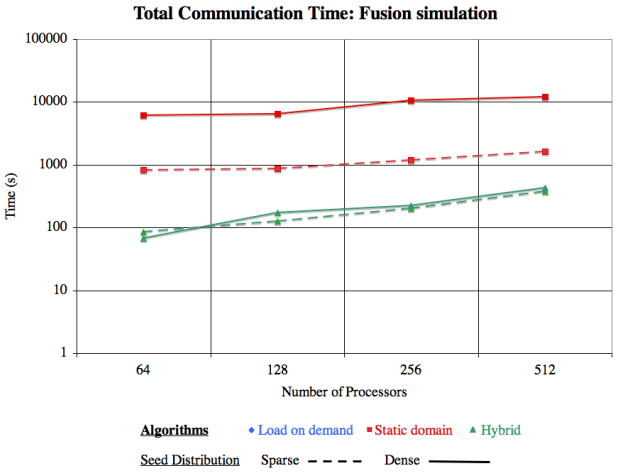

Figure 11: A logarithmic scale plot of total time spent performing communication for all three algorithms, as applied to the fusion dataset with both sparse and densely placed seed points.

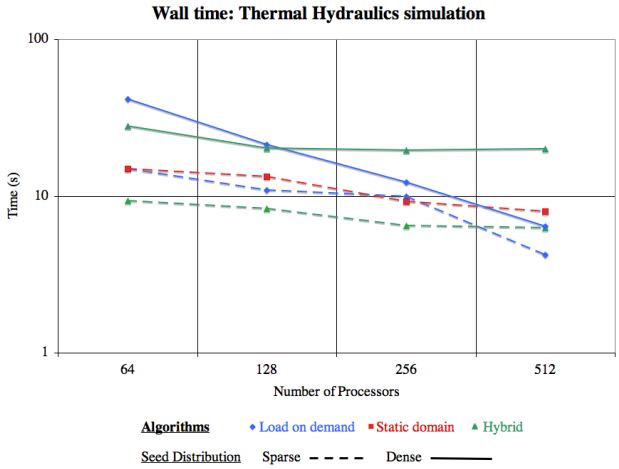

Figure 13: A logarithmic scale plot of wall clock time for all three algorithms, as applied to the thermal hydraulics dataset with both sparse and dense initial seed points.

This algorithm is well suited to datasets were I/O is expensive and seed point sets and flow that distributes streamline computation uniformly throughout the dataset. However, this algorithm can perform very poorly if the streamline distribution is not uniform, e.g. a flow with sources and sinks. In this case, the streamline integration will be largely focused on the few processors that contain the blocks with the sources and sinks.

Hybrid Master/Slave was designed to parallelize across both streamlines and blocks and to dynamically adapt the parallelization strategies as the streamline computation evolves. Our tests cases indicate that this algorithm is best suited for a wide variety of situations and is the recommended algorithm to use for general purpose parallel streamline computation. It is particularly recommended when the flow field is not well understood, as Hybrid Master/Slave will dynamically adapt to the specific nature of the flow. Once the nature of the flow is well understood, the Static Allocation or Load On Demand algorithms are suggested, if they are able to optimize their strengths.

7. SUMMARY

In this paper we undertook a performance characterization of two algorithms based on established streamline parallelization approaches, and one novel algorithm for the generation of streamlines for flow analysis of large datasets. Our results demonstrate fundamental scalability limitations of the Static Allocation and Load On Demand algorithms for

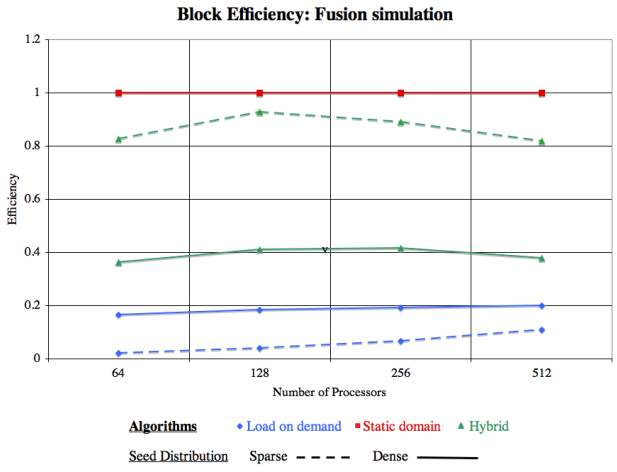

Figure 12: Block efficiency for all three algorithms, as applied to the fusion dataset with both sparse and densely paced seed points.

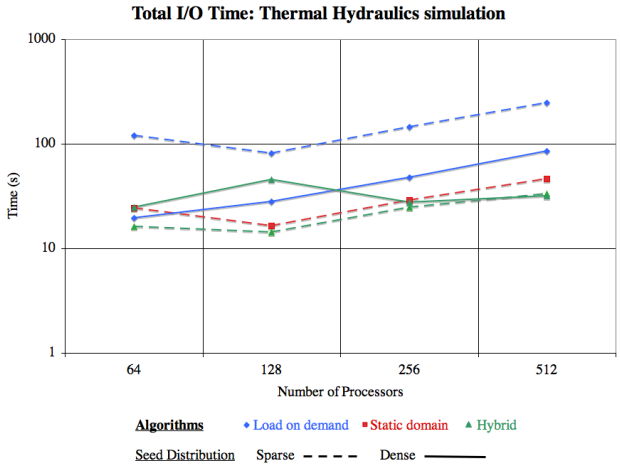

Figure 14: A logarithmic scale plot of total I/O time for all three algorithms, as applied to the thermal hydraulics dataset with both sparse and dense initial seed points.

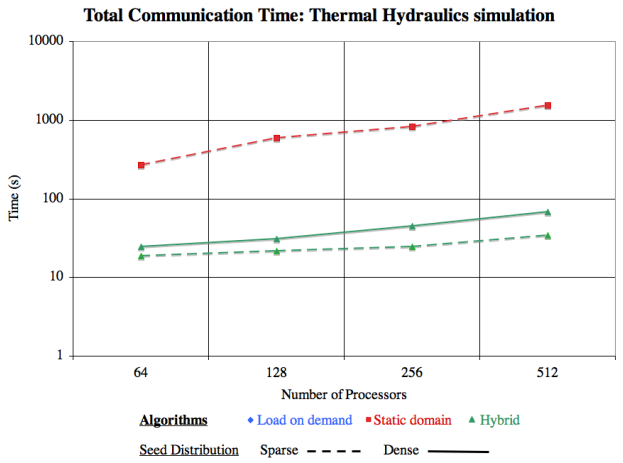

Figure 15: A logarithmic scale plot of total communication time for all three algorithms, as applied to the thermal hydraulics dataset with both sparse and dense initial seed points.

specific application problems due to memory and processing constraints. This is a consequence of the fact that both algorithms parallelize only over data size or streamline count, respectively.

We demonstrated that a hybrid approach that adapts to the system resources and the unknown flow characteristics provides good scalability for a representative set of large datasets. We contribute a new heuristic-based algorithm that scales over both data size and seed set size by balancing I/O, communication, and computation to achieve scalability on large numbers of processors and for very large data.

While our hybrid algorithm typically performs better than

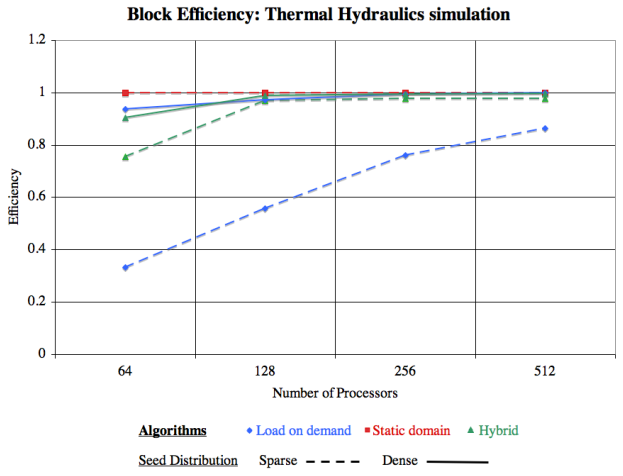

Figure 16: Block efficiency for all three algorithms, as applied to the thermal hydraulics dataset with both sparse and densely paced seed points.

than the other two algorithms, there are circumstances in which Static Allocation and Load On Demand algorithms can be superior to the hybrid algorithm, mostly due to the exploitation of efficiencies in communication.

8. FUTURE WORK

Going forward with this work, we plan to further improve our hybrid parallelization scheme to further address load imbalances. Distributing the work is based on several heuristics that may be more or less appropriate depending on data set properties. One possibility for improving scaling behavior would be observing communication and processor utilization patterns and modify used heuristics based on these indicators. Specifically, we have found that processor starvation is often a limitation to large scalability that we believe can be addressed through additional heuristics.

Our current study examines in detail the performance of streamline computation for large-scale data sets. The same considerations also apply to pathlines, which depend on considerably larger amounts of data since it becomes necessary to advance through multiple time steps of a simulation as well as space. We have performed preliminary studies that suggest that, similar to streamlines computation, minimizing redundant I/O remains a major challenge. In particular, computing pathlines leads to many small reads that can often overwhelm the file system and dramatically affect scalability. We intend to explore reading a block from disk only once and communicating it in the same way as streamlines are passed around in our current implementation and what impact this would have on performance. Furthermore, we plan to develop improved caching strategies for stream- and pathlines. One topic of interest is whether some limited pre-processing (e.g., binary clustering used by Hu et al. [24]) can be used to gain a sufficient overview over streamline behavior to make an informed decision concerning distributing blocks to streamlines.

Another important research area is considering algorithms that do not depend on an a priori knowledge of all seed points, but add new seed points dynamically based on an ongoing streamline calculation. One application area where this becomes necessary is the calculation of stream surfaces. It would be interesting to study how these additional seed points affect load balancing, in particular since it may be possible to base "educated guesses" on local streamline behavior. In principle, our architecture should be suited to the dynamic creation of streamlines with few modifications.

Communicating streamline geometry accounts for a large proportion of communication cost observed in our studies, particularly when following streamlines for a long period of time. In many streamline applications (e.g. Poincaré puncture plots) the total streamline geometry is not of interest in future integration. In these classes of problems, it should be sufficient to communicate solver state as well as some relatively compact derived quantities. It would be interesting to determine how these properties may be used to reduce communication overhead and how doing so affects the relative performance of our distribution strategies.

9. ACKNOWLEDGMENTS

This work was funded in part by the SciDAC2 Visualization and Analytics Center for Enabling Technologies and ASCR's Visualization Base Program by the Director, Office of Science, Office of Advanced Scientific Computing Research, of the U.S. Department of Energy under Contract No. DE-AC03-76SF00098.

10. REFERENCES

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

[2] B. G. Becker, N. L. Max, and D. A. Lane. Unsteady flow volumes. In *IEEE Visualization*, pages 329–, 1995.

[3] R. Bruckschen, F. Kuester, B. Hamann, and K. I. Joy. Real-time out-of-core visualization of particle traces. In *Proceedings of the IEEE Symposium on parallel and large-data visualization and graphics (PVG)*, pages 45–50, Piscataway, NJ, USA, 2001. IEEE Press.

[4] B. Cabral and L. C. Leedom. Highly parallel vector visualization using line integral concolution. In *Proc. SIAM PPSC '95*, pages 802–807, 1995.

[5] C. Y. Cardall, A. O. Razoumov, E. Endeve, E. J. Lentz, and A. Mezzacappa. Toward five-dimensional core-collapse supernova simulations. *Journal of Physics: Conference Series*, 16:390–394, 2005.

[6] L. Chen and I. Fujishiro. Optimizing parallel performance of streamline visualization for large distributed flow datasets. In *Proc. IEEE VGTC Pacific Visualization Symposium 2008*, pages 87–94, 2008.

[7] D. Ellsworth, B. Green, and P. Moran. Interactive terascale particle visualization. In *Proc. IEEE Visualization*, pages 353–360, Washington, DC, USA, 2004. IEEE Computer Society.

[8] 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.

[9] P. Fischer, J. Lottes, D. Pointer, and A. Siegel. Petascale algorithms for reactor hydrodynamics. *Journal of Physics: Conference Series*, 125:1–5, 2008.

[10] C. Garth, H. Krishnan, X. Tricoche, T. Bobach, and K. I. Joy. Generation of accurate integral surfaces in time-dependent vector fields. *IEEE Transactions on Visualization and Computer Graphics*, 14(6):1404–1411, 2008 Nov-Dec.

[11] E. Hairer, S. P. Nørsett, and G. Wanner. *Solving Ordinary Differential Equations I, second edition*, volume 8 of *Springer Series in Comput. Mathematics*. Springer-Verlag, 1993.

[12] G. Haller and G. Yuan. Lagrangian coherent structures and mixing in two-dimensional turbulence. *Physica D*, 147:352–370, 2000.

[13] J. P. M. Hultquist. Constructing stream surfaces in steady 3d vector fields. In A. E. Kaufman and G. M. Nielson, editors, *Proceedings of IEEE Visualization 1992*, pages 171 – 178, Boston, MA, 1992.

[14] D. A. Lane. UFAT – A Particle Tracer for Time-Dependent Flow Fields. In *Proc. IEEE Visualization '94*, pages 257–264, 1994.

[15] M. Mathur, G. Haller, T. Peacock, J. Ruppert-Felsot, and H. Swinney. Uncovering the lagrangian skeleton of turbulence. *Phys. Rev. Lett.*, submitted, 2006.

[16] T. McLoughlin, R. S. Laramee, R. Peikert, F. H. Post, and M. Chen. Over Two Decades of Integration-Based, Geometric Flow Visualization. In M. Pauly and G. Greiner, editors, *Eurographics STAR - State of The Art Report (to appear)*, April 2009.

[17] S. Muraki, E. B. Lum, K.-L. Ma, M. Ogata, and X. Liu. A pc cluster system for simultaneous interactive volumetric modeling and visualization. In *Proc. IEEE Symp. on Parallel and Large-Data Visualization and Graphics (PVG)*, page 13, Washington, DC, USA, 2003. IEEE Computer Society.

[18] P. J. Prince and J. R. Dormand. High order embedded runge-kutta formulae. *Journal of Computational and Applied Mathematics*, 7(1), 1981.

[19] S. Shadden, J. Dabiri, and J. Marsden. Lagrangian analysis of fluid transport in empirical vortex ring flows. *Physics of Fluids*, 18:047105, 2006.

[20] C. Sovinec, A. Glasser, T. Gianakon, D. Barnes, R. Nebel, S. Kruger, S. Plimpton, A. Tarditi, M. Chu, and the NIMROD Team. Nonlinear magnetohydrodynamics with high-order finite elements. *J. Comp. Phys.*, 195:355, 2004.

[21] D. Sujudi and R. Haimes. Integration of particles and streamlines in a spatially-decomposed computation. In *Proc. Parallel Computational Fluid Dynamics*, Los Alamitos, CA, 1996. IEEE Computer Society Press.

[22] S.-K. Ueng, C. Sikorski, and K.-L. Ma. Out-of-core streamline visualization on large unstructured meshes. *IEEE Transactions on Visualization and Computer Graphics*, 3(4):370–380, 1997.

[23] D. Xue, C. Zhang, and R. Crawfis. Rendering implicit flow volumes. In *Proc. IEEE Visualization '04 Conference*, pages 99–106, 2004.

[24] H. Yu, C. Wang, and K.-L. Ma. Parallel hierarchical visualization of large time-varying 3d vector fields. In *Proc. Supercomputing 2007*, 2007.