Parallel Stream Surface Computation for Large Data Sets

David Camp
University of California, Davis
Lawrence Berkeley National Laboratory

Hank Childs
Lawrence Berkeley National Laboratory
University of California, Davis

Christoph Garth
University of Kaiserslautern

David Pugmire
Oak Ridge National Laboratory

Kenneth I. Joy
University of California, Davis

Acknowledgements: This research used resources of the National Energy Research Scientific Computing Center (NERSC), which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231. This work was also supported by the Director, Office of Advanced Scientific Computing Research, Office of Science, of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231 through the Scientific Discovery through Advanced Computing (SciDAC) program's Visualization and Analytics Center for Enabling Technologies (VACET).

Parallel Stream Surface Computation for Large Data Sets

David Camp*
University of California, Davis
Lawrence Berkeley National Laboratory

Hank Childs†
Lawrence Berkeley National Laboratory
University of California, Davis

Christoph Garth‡
University of Kaiserslautern

David Pugmire§
Oak Ridge National Laboratory

Kenneth I. Joy¶
University of California, Davis

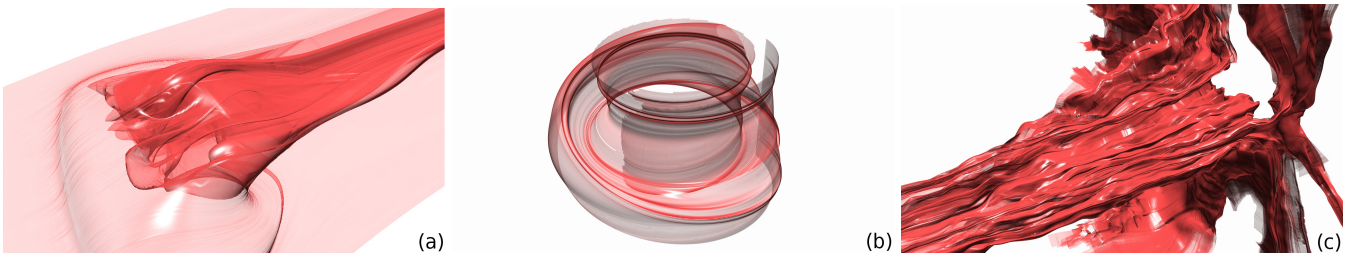

Figure 1: Stream surfaces generated during this study. On the left, flow around an ellipsoid. In the middle, flow in a tokamak. On the right, flow from a thermal hydraulics simulation.

Abstract

Parallel stream surface calculation, while highly related to other particle advection-based techniques such as streamlines, has its own unique characteristics that merit independent study. Specifically, stream surfaces require new integral curves to be added continuously during execution to ensure surface quality and accuracy; performance can be improved by specifically accounting for these additional particles. We present an algorithm for generating stream surfaces in a distributed-memory parallel setting. The algorithm incorporates multiple schemes for parallelizing particle advection and we study which schemes work best. Further, we explore speculative calculation and how it can improve overall performance. In total, this study informs the efficient calculation of stream surfaces in parallel for large data sets, based on existing integral curve functionality.

Index Terms: D.1.3 [Programming Techniques]: Concurrent Programming—Parallel programming; Computer Graphics [I.3.3]: Picture/Image Generation—Display algorithms

1 Introduction

A stream surface is an infinite set of integral curves originating from a seeding curve. It is a powerful visualization tool for insight into characteristics and features of vector fields. In practice, stream surfaces are approximated by the triangulation of adjacent pairs of integral curves to create the surface. Typically, the stream surface is constructed adaptively. Particles are placed along the seeding curve and advected, creating integral curves. When two adjacent integral curves diverge too far, new particles are inserted to adapt the computation and guarantee the accuracy of the approximation. For maximum surface accuray, especially for long integration times, these new particles must be placed on the initial seeding curve. Unfortunately, this is often not efficient, in a distributed-memory parallel environment, because the region containing the new particles is often not readily available to perform further integration.

The performance characteristics of stream surfaces are different than streamlines and other particle advection-based techniques. First, all particles originate along a seeding curve, which heavily emphasizes certain regions of the volume, namely the regions that contain the seeding curve and those in close proximity. Further, the addition of "refinement" particles inserted back on the seeding curve results in a fundamentally different access pattern. In this study, we ask the question: what is the most efficient way to carry out a stream surface calculation on large data, given that its access patterns are different than other particle advection algorithms?

An important start to our work is parallelizing the stream surface calculation. Parallelization is necessary for stream surfaces that require more memory than is available on a desktop machine and also to reduce the time spent calculating the surface. This parallelization process goes beyond advecting particles in parallel. We must detect when adjacent integral curves have diverged too far, diminishing the quality of the surface approximation, even when those integral curves reside on different processing elements. Further, we must be able to create the final surface in a way that guarantees that no processing element exceeds its available memory.

The algorithm we developed supports different approaches for parallelizing particle advection. Since every particle originates on the seeding curve – which occupies a relatively small part of the overall volume – efficient parallelization is difficult. We study two approaches, parallelizing-over-particle and parallelizing-over-data, and their relative merits in this environment.

Finally, we consider the idea of speculative refinement. The motivating idea is that advecting particles is less expensive when the region of the mesh a particle traverses is already loaded from disk and in primary cache. Clearly, it would be ideal to know *a priori* which particles provide a good approximation of the surface. But this is not possible, since the data-dependent nature of the algorithm means that separation of adjacent integral curves must be detected as the particles advect. In short, speculative calculation will advect more particles, but the cost of advecting each particle will be reduced. With this study, we endeavored to assess this balance qualitatively: under what circumstances does speculative calculation of

*e-mail:dcamp@lbl.gov
†e-mail:hchilds@lbl.gov
‡e-mail:garth@cs.uni-kl.de
§e-mail:pugmire@ornl.gov
¶e-mail:joy@cs.ucdavis.edu

particles improve overall performance? When does speculative calculation stop being beneficial?

Parallel particle advection performance is highly dependent on the initial configuration – where the seeding curve lies for stream surfaces – and the nature of the velocity field. We designed a study that covers a variety of configurations and determine when the parallelization methods and speculative computation perform best. Specifically, we used three different data sets and varied the size of the seeding curve, the integration time (which affects surface length), and the amount of speculative calculation. The combination of these variables resulted in 120 different experiments.

The contributions of this paper are:

- The description of an algorithm for calculating stream surfaces on a distributed-memory parallel system;
- Analysis of two different parallel particle advection algorithms, parallelizing-over-particle and parallelizing-over-data, with respect to stream surfaces, to understand their performance differences;
- The concept of speculative refinement in the context of stream surfaces, and analysis of its benefits; and
- A thorough set of experiments that cover a variety of stream surface workloads.

2 Background and Previous Work

2.1 Stream Surface Definition and Computation

In the following, we assume that $v(x)$ is a three-dimensional vector field, defined over a finite domain $\Omega \subset R^3$. In this paper, v is assumed stationary, *i.e.* constant in time. An *integral curve* $S(x_0)$ of v is the solution to the ordinary differential equation

$$\dot{S}_{x_0}(t) \;=\; v(S(t)) \tag{1}$$

with initial condition

$$S_{x_0}(0) \;=\; x_0 \tag{2}$$

Simply put, S is a curve that contains the point x_0 and is tangent to the vector field at every point over time. The intuitive understanding associated with integral curves is that of the trajectories of massless particles that are advected through a domain by v. Furthermore, the point x_0 from which the particle is initially released is called the *seed point* or simply *seed*, and one calls an integral curve *forward* if $t > 0$ and *backward* if $t < 0$, respectively. In the typical case found in visualization where v is given in discrete form (*e.g.*, as an interpolated variable over a regular or unstructured mesh), S can be approximated using numerical integration techniques (cf. [11]).

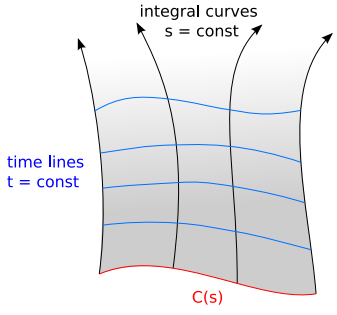

Figure 2: Stream surfaces consist of families of particle trajectories emanating from the curve $C(s)$. Alternatively, they can be viewed as the surface traversed by C under advection in the vector field; instantaneous snapshots of C are called time lines.

If C is a one-dimensional curve, contained in Ω and parameterized by s, a *stream surface* $\mathscr{S}$ is defined by

$$\mathscr{S}(s,t) \;:=\; S_{C(s)}(t).$$

In words, $\mathscr{S}$ is the union or continuum of integral curves originating on the *seeding curve* C. While $\mathscr{S}(s,\cdot)$ coincides with an individual integral curve, the surface lines given by $\mathscr{S}(\cdot,t)$ are called *time lines*. Figure 2 provides a graphical explanation of these terms.

Observe that $\mathscr{S}$ has a natural parameterization in the form of (s,t) coordinates. Every point on the surface, given s and t, can be computed by propagating the integral curve starting at $C(s)$ through the application of numerical integration until it reaches t. The goal of any stream surface algorithm is to construct a geometric approximation of $\mathscr{S}$, to be visualized, using a sparse set of integral curves.

The need to generate stream surfaces efficiently, *i.e.* using a minimum number of integral curves, was first recognized by Hultquist [12] in 1992. He was the first to describe surface integration in terms of an advancing front. In his approach, the starting curve is discretized using a finite set of points that seed integral curves to form a skeleton of the stream surface. The integral curves are individually advanced by a single integration step at a time, resulting in a sequence of points for each integral curve called an *advancing front*. Here, Hultquist advances integral curves according to a clever heuristic that keeps the angles between adjacent front segments small. Adjacent pairs of integral curves form *ribbons*, and triangulation is performed per-ribbon by a shortest-diagonal heuristic.

Due to the strong deformations a stream surface exhibits, resulting from complex flows, using a fixed (even if high) number of curves to represent the surface yields inadequate results. Hultquist employed *adaptive refinement* by using the distance between adjacent integral curves to control the front resolution through insertion and removal of integral curves. Furthermore, additional criteria address the fact that stream surfaces may "tear" (*i.e.*, become discontinuous) at object boundaries or near saddle-like structures. Here, ribbons where this behavior is identified are not propagated further, and a single front is split into multiple fronts.

Overall, Hultquist's method performs well for moderately complex flows, but does not cope well with folding, shearing or twisting of stream surfaces, since the refinement strategy does not take these issues into account. Hultquist's algorithm was augmented by Stalling [23] by incorporating local topological information into the triangulation process, which addresses the strong deformation found near critical points, and slightly modifying the refinement criteria to address some of the cases mentioned above. Garth et al. [10] introduced several modifications to Hultquist's original scheme to allow stream surface computation in the presence of very complex flow structures. By moving from time-based integral curve integration to arc length-based integration and incorporating front refinement criteria based on curvature, better resolution control is achieved, and complex flow patterns can be adequately resolved. In a later paper, these authors described an algorithm that translates these results to time-varying flow [9].

In contrast to methods that compute the surface geometry explicitly as a triangle mesh, other methods avoid integral curve computation. van Wijk [27] gave a global approach that implicitly represents stream surfaces as implicit surfaces in an advected scalar field. Scheuermann et al. [21] exploited the existence of an analytic flow solution for tetrahedral grids with linear interpolation to propagate a stream surface through tetrahedra basis. While these techniques are conceptually elegant, the restrictions introduced with respect to choice of starting curve or surface resolution severely limit the flexibility of stream surfaces for vector field visualization purposes. Aside from explicit construction of stream surface geometry, methods exist that create the visual impression of a stream surface by using particles, *e.g.*, [26]. While simple to implement and broadly

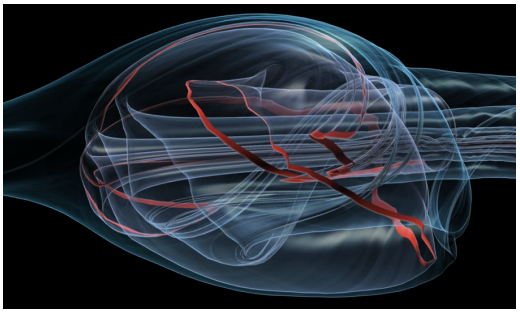

Figure 3: A vortex breakdown bubble is visualized by a single complex stream surface rendered using illustrative rendering techniques. Image reproduced from [13] with permission.

applicable, the visual clarity of these schemes is often lacking, as the depth-enhancing quality of shaded surfaces is lost.

Advanced visualization of integral surfaces was introduced by Löffelmann [15] in proposing texture mapping of arrows on stream surfaces (*stream arrows*) to convey the local direction of the flow, and Stalling [23] employed a LIC method towards the same goal. Garth et al. [9], Hummel et al. [13], and Born et al. [2] recently investigated advanced rendering techniques including non-photorealistic rendering and better texture mapping to achieve improved visual depictions of stream surfaces (cf. Figure 3). These examples document the potential of stream surfaces over simpler methods, such as streamlines, to fully elucidate the complex nature of flow in three-dimensional vector fields. Furthermore, stream surfaces have also been used as building blocks for more complex visualization techniques, *e.g.*, in an algorithm for the extraction of vortices from CFD data [10].

2.2 Parallel Considerations

Existing stream surface algorithms are not easily parallelized. For example, Hultquist's method and its generalizations [12, 23, 10] crucially rely on single-stepping individual integral curves and applying heuristics after each such step to guarantee viable results; thus, parallel computation is difficult to employ for such algorithms. Furthermore, the entire vector field must be stored in memory to achieve reasonable performance.

Although we are aware of no previous work for parallel stream surface computation over large data sets, there has been considerable previous work in the area of parallel integral curves and other integral curve techniques. Generally, both the data set, represented as a number of disjoint blocks, and the computation in the form of integration work, can be distributed. An early treatment of the topic was given by Sujudi and Haimes [24], who made use of distributed computation by assigning each processor one data set block. An integral curve is communicated among processors as it traverses different blocks. Other examples of applying parallel computation to integral curve-based visualization include the use of multiprocessor workstations to parallelize integral curve computation (*e.g.*, [14]), and research efforts focusing on accelerating specific visualization techniques [4]. Similarly, PC cluster systems were leveraged to accelerate advanced integration-based visualization algorithms, such as time-varying Line Integral Convolution (LIC) volumes [16] 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, Ueng et al. [25] presented a technique to compute integral curves in large unstructured grids using an octree partitioning of the vector field data for fast fetching during integral curve construction using a small memory footprint. Taking a different approach, Bruckschen et al. [3] described 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.

Different partitioning methods were introduced with the aim of optimizing parallel integral curve computation. Yu et al. [28] 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 integral curves for data analysis as opposed to obtaining a qualitative data overview. Chen and Fujishiro [5] applied a spectral decomposition using a vector-field derived anisotropic differential operator to achieve a similar goal with similar drawbacks. Peterka et al. [18] did a study of parallel particle tracing for steady-state and time-varying flow fields in which they configured the 4D domain decomposition into spatial and temporal blocks that combine in-core and out-of-core execution in a flexible way that favors faster run time or smaller memory usage. They also compared static and dynamic partitioning approaches. They learned several lessons for steady flow, including that computational load balance is critical at smaller system scale. Beyond that, communication is the primary bottleneck. They also found that round robin domain decomposition remains the best layout for load balancing. Finally, Nouanesengsy et al. [17] employed an algorithm that preprocessed the data, analyzed the probability that particles would move from one block to another, and made decisions about where to load blocks (sometimes redundantly) based on this analysis. While impressive results were achieved for streamlines, their algorithm likely would need to be specifically adapted for stream surfaces, since the probabilities of particles moving from block to block are no longer independent as all particles originate on a seeding curve.

Most relevant to this paper, Pugmire et al. presented a systematic study of the performance and scalability of three parallelization algorithms [20]: the first two correspond to parallelization-over-particle 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. The parallelization-over-particle and over-data blocks techniques are discussed further in section 4. The intent of this work is to implement stream surface computation in terms of the above integral curve framework and investigate the performance and feasibility of different parallelization schemes in this context.

3 Adaptive Integration-Based Algorithms

In the following section, we describe a general technique for the approximation of integral surfaces. The surface approximation is achieved by the successive approximation of integral curves. We first describe an integral curve refinement algorithm to create the surface skeleton and then use that skeleton to create the surface approximation.

3.1 Surface Refinement

The most important problem in the stream surface algorithm is the refinement strategy. The refinement strategy must handle the presence of discontinuities in the surface. To do this we specify the following constants: the refinement distance D_{refine}, the minimum distance between adjacent integral curves to achieve surface quality, and the discontinuity distance $D_{discont}$, the distance at which we

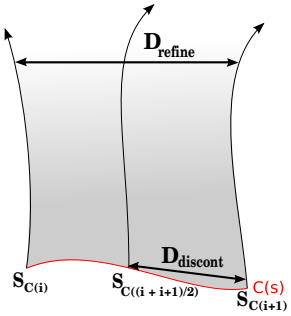

Figure 4: If the distance between streamlines ever exceeds the D_{refine} distance we must refine the surface between $S_{C(i)}$ and $S_{C(i+1)}$. The $D_{discont}$ distance is the minimum distance between streamlines inserted on the seed curve.

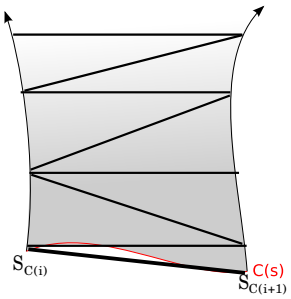

Figure 5: Adjacent pairs of integral curves form *ribbons* [12]. Triangulation is performed per-ribbon by a shortest-diagonal heuristic. This figure shows a sample triangulation between $S_{C(i)}$ and $S_{C(i+1)}$.

stop the refinement because of discontinuities. We next describe an algorithm for adaptive refinement of the surface approximation done by incremental insertion of new integral curves S into the sequence of curves $S_{C(i)}, i = 0, \ldots, N$ originating at the seeding curve C. We assume the curve C is a piecewise linear curve. Surface refinement is needed if the distance between $S_{C(i)}$ and $S_{C(i+1)}$ integral curves ever exceeds the distance D_{refine} along the length of the both curves, see Figure 4 for an example distance. To refine the surface, we insert a new seed point on the seeding curve between $S_{C(i)}$ and $S_{C(i+1)}$ seed points to increase the surface definition. There could be discontinuities in the stream surface that will cause the refinement to never stop trying to insert more integral curves to fill this distance. To handle the presence of discontinuities, the refinement is stopped if the distance between $S_{C(i)}$ and $S_{C(i+1)}$ seed points is less than $D_{discont}$, see Figure 4 for an example distance. This algorithm refines the integral curve sequence until a certain level of quality is achieved by the distance measurements.

We formulate the following iterative refinement algorithm:

1. Let $S_{C(i)}(), i = 0, \ldots, N$ be a set of integral curves created from an initially provided set of seed points and create an empty set S_{insert} to contain inserted integral curve seeds points from the refinement process.
2. For each adjacent integral curve set $(S_{C(i)}, S_{C(i+1)})$ compute the distance between them; if that distance is ever greater than D_{refine}, create an integral curve seed point S_{refine} at the midpoint between the seed points $S_{C(i)}$ and $S_{C(i+1)}$.
3. If the distance between the seed points $S_{C(i)}$ and $S_{C(i+1)}$ is greater than $D_{discont}$, add S_{refine} into S_{insert}.
4. If S_{insert}, is empty then calculate stream surface.
5. Otherwise, compute the integral curves in the S_{insert} set and then merge the integral curves into the $S_{C(i)}$ set. The merge is needed to continue the refinement check, so continue at step 2.

3.2 Stream Surface Generation

The above refinement algorithm will build a set of integral curves propagated from the seeding curve, resulting in a skeleton of the surface. Typically, for each curve the numerical integration algorithm outputs a sequence of points that are used to represent the integral curve in a piecewise linear fashion. We generate a triangulation from this set of integral curves by grouping them into ribbons and triangulating each ribbon using a shortest diagonal approach, see Figure 5 for an example. The ribbons are created by examining the two diagonals connecting the active points and next points on both curves and taking the shorter one as the new active edge, simultaneously generating a triangle formed by the three points. The active point is moved to the top point of the triangle and shortest diagonal is calculated again. The ribbon terminates if the active edge moves beyond the last point of either integral curve. The pair of integral curves may not contain the same number of points and will end at the lesser point list. This algorithm generates a triangle-based representation of the integral surface approximation of S_C, which can then be directly rendered to achieve a visual representation of the approximated integral surface. While it is possible to post-process the generated triangle mesh to reduce its size for faster rendering, we have not explored this issue as it is outside the scope of this project.

4 Parallel Adaptive Stream Surface Algorithm

4.1 Parallel Advecting Algorithms

There are two straightforward approaches to partitioning the computational workload of generating integral curves by advecting particles through a vector field: parallelization with respect to the particles, and parallelization with respect to the data. In this section, we give an overview of our parallelization algorithms, along with the suitability of each to particular uses cases. Further, Figure 6 illustrates each algorithm for a simple case.

Parallelize-over-Particle (POP) The POP algorithm parallelizes over a set of particles, assigning each message passing interface (MPI) task a fixed number of particles from which integral curves are then integrated. Data blocks are loaded on demand when required, *i.e.* when integral curve 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 integral curve can be continued on the current set of resident blocks. Since blocks might be used repeatedly during the integration of integral curves, they are kept in the cache as long as possible. Data blocks are evicted in least recently used order to make room for new blocks. This algorithm benefits from a small amount of communication; the only communication occurs at initialization and termination.

The initial assignment of particles to tasks is based on spatial proximity, following the reasoning that the integral curves traced from spatially proximate seeds are likely to traverse the same regions, and thus blocks, of a data set.

This parallelization strategy tends to balance the work load over the set of tasks since the particles are uniformly distributed and this also leads to a nice balance of memory usage between tasks. However, since I/O is performed as needed, I/O costs can dominate performance based on the nature of the seeding conditions, and the vector field. This technique can be attractive to stream surfaces since very dense seeding is required along the seeding curve.

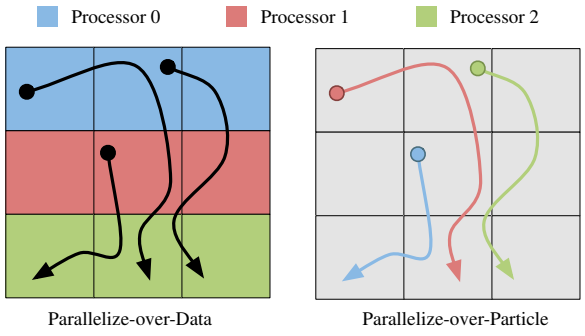

Figure 6: Parallelization approaches for distributed integral curve integration. Parallelize-over-Data relies on a fixed assignment of data blocks to processors and passes integral curves between processor as they advance into new blocks. Parallelize-over-Particle distributed integral curves evenly among processors, and data blocks are loaded on demand when required to integrate an integral curve.

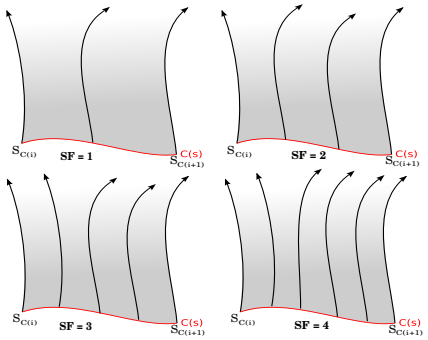

Figure 7: The speculative factor SF is used to increase the amount of refinement done between $S_{C(i)}$ and $S_{C(i+1)}$ integral curves. This figure shows an example of four different SF values. There is tension between adaptivity (compute incrementally and as little as possible) and parallelization (compute in large batches). The SF is designed to balance these tensions.

Parallelize-over-Data (POD) The POD approach distributes data blocks over MPI tasks using a fixed assignment. Integral curves 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.

After each integral curve is advected as far as possible using the available data blocks, the next block is determined and the integral curve is sent to the corresponding task; then, further integral curves are received from other tasks and stored for integration. Integral curve communication is limited to a small amount of integration state. Integral curve geometry generated during the integration remains with the task that generated it, and is reassembled into a complete integral curve once computation is complete. The parallelize-over-data 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.

This parallelization strategy has the advantage of performing minimal I/O, which can be a determining factor in algorithm performance. However, the algorithm is subject to work imbalance as the task that owns a data block is responsible for computing every integral curve which passes through it. We expect in the case of stream surfaces, many integral curves will be created very close together which could cause high load imbalance, depending on the data layout. This could also lead to memory imbalance between tasks because each task keeps track of all integral curves segments created on its data.

4.2 Stream Surface Algorithm

Our stream surface algorithm treats the parallel advecting algorithm as a "black box," which is beneficial in that any parallelization strategy can be used to calculate the integral curves. This also is beneficial in the development of new parallel advecting algorithms as they are easily integrated into the stream surface algorithm. But this abstraction also has an unknown penalty because of the lack of coordination between the stream surface and advection algorithm. There is a tension between these algorithms; the stream surface algorithm wants to compute incrementally and as little as possible, and the advection algorithms want to parallelize large amounts of work. Speculative calculation is explored in this paper as a way to reduce this tension. The speculative calculation is done by increasing the amount of refinement done between the two consecutive integral curves. Where the normal refinement inserts one new integral curve, we can insert more than one integral curve per refinement, and call this speculative factor SF. Figure 7 has examples of four different speculative factors. Again, the speculative factor will be used to balance the tension between the amount of work that the stream surface and parallel advection algorithms want to do.

```
Mesh Calculate_Stream_Surface( C, CP )
{
  S_C = Calculate_Initial_Set_Of_Integral_Curves( C, CP );
  S_C = Distribute_Integral_Curves( S_C );
  while( S_insert = Adaptive_Refinement_Check(S_C) )
  {
    S_refine = Calculate_New_Integral_Curves( S_insert );
    S_C = Distribute_New_Integral_Curves( S_refine );
  }
  return Create_Integral_Curve_Surface_Mesh( S_C );
}
```

Listing 1: Stream Surface Algorithm

We assume the user will provide the piecewise linear seeding curve C and an initial set of seed points CP. Using these settings the parallel advection algorithm, POP or POD, will calculate the initial set of integral curves S_C. After the integral curves are generated, they are evenly distributed across tasks. Each task will contain a subset of the stream surface. The first integral curve will be duplicated between adjacent tasks to cover the space between surface segments. This obviates the need for parallel coordination for refinement checks between adjacent tasks. Dividing the integral curves evenly between tasks will distribute the memory requirement evenly between tasks. Each task will do an adaptive refinement check to determine if the surface quality is good or if more integral curves are needed to refine the surface to the required resolution. The surface quality is determined by checking the distance between two consecutive integral curves along the entire length of the curve. The user will specify a minimum distance D_{refine} to determine the surface quality. If the requirement is not met the algorithm will refine between the integral curves. Refinement is done by adding a number of new seed points, specified by the speculative factor, on the seeding curve between the $S_{C(i)}$ and $S_{C(i+1)}$ integral curves. If the surface has discontinuities, the adaptive refinement step will never satisfy the minimum distance D_{refine} between integral curves. To stop the adaptive refinement process, we will stop inserting new seed points if the distance between the $S_{C(i)}$ and $S_{C(i+1)}$ seed points is less than $D_{discont}$. After the adaptive re-

finement check we will have a set of seed points that need to be calculated.

The new seed points are communicated to the processing task – which task depends on the advection algorithm – and then they are integrated. The new integral curves must be communicated to each task that requested the refinement and added to the S_C set. This is needed to check if more surface refinement is needed. This process is repeated until the adaptive refinement check is satisfied.

After resolution of the surface is adequate, each task generates triangle strips with the integral curves it contains and renders the stream surface. The triangle strips are generated between two consecutive integral curves using the integral curve segments. The first integral curve in a task is duplicated between adjacent tasks to create the triangle strip needed to connect the surface between tasks. The rendering is done in parallel because of the large number of triangles generated. This surface then will allow for the application of surface shading techniques to complete the visualization.

5 Experiments

5.1 Test Cases

We set up tests to cover a wide range of computational workloads, varying over starting seeding curve size (wide or narrow), integration time (long or short), and speculative factor (1–6) run for each of three different data sets. This resulted in 120 different test combinations, e.g. "narrow-short-speculative factor=1" test. The data sets are:

Fusion: This data set is from a simulation of magnetically confined fusion in a tokamak device. To achieve stable plasma equilibrium, the field lines of the magnetic field need to travel around the torus in a helical fashion. Using stream surfaces the scientist can visualize this magnetic field. The simulation was performed using the NIMROD code [22]. This data set has the unusual property that most integral curves are approximately closed and traverse the torus-shaped vector field domain repeatedly which stresses the data cache. All tests start with 128 seed points and they integrate for 3 and 6 time units, respectively, for the short and long times. The short time will complete one rotation of the magnetic field while the long time will complete two rotations. Figure 1b is an illustrative rendering of the surface.

Thermal Hydraulics: In this data set, twin inlets pump air into a box, with a temperature difference between the inlets. The air mixes in the box and exits through an outlet. Mixing of "hot" and "cold" air, residence time in the box, and identification of both stagnant and highly turbulent regions are areas of active study. The simulation was performed using the NEK5000 code [8]. All tests start with 361 seed points and they integrate for 5 and 10 time units, respectively, for short and long times. The seed points are placed around one of the inlets. When advecting, they travel to the top of the tank and then expand out. Figure 1c is an illustrative rendering of the top part of the surface.

Ellipsoid: This data set is from an unsteady simulation of the flow around an ellipsoid, where the angle of the surrounding flow changes over time. Vortex shedding can be observed on the ellipsoid boundary. All tests start with 128 seed points and they integrate for 10 time units for long time. We did not use a short time test because all of the action in this data set occurs on the other side of the ellipsoid object. Figure 1a is the surface created from our seeding curve.

Although our techniques are readily applicable to any mesh type and decomposition scheme, we wanted to consider large data sets, so the data sets we study here were resampled to a regular mesh of 512 blocks with 1 million cells per block. 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.

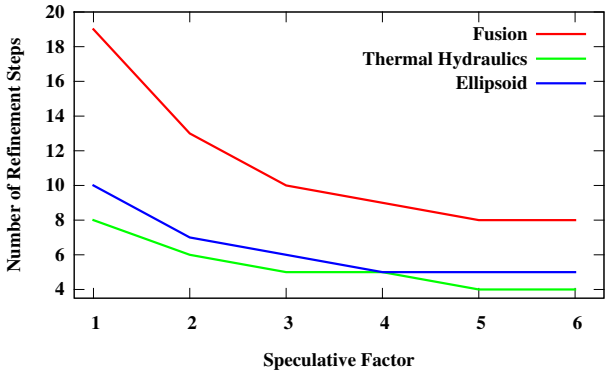

Figure 8: As the speculative factor increases, the number of refinement steps drops. This can reduce the busy wait time and the number of data blocks that need to be loaded.

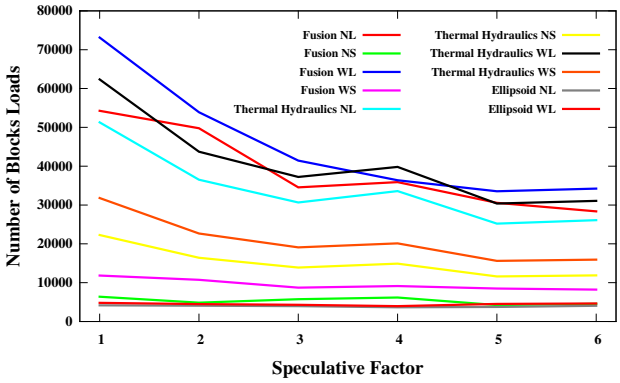

Figure 9: As the speculative factor increases, the number of data block loads drops. This can reduce the I/O time.

In this work we use the 5th-order Runge-Kutta implementation given by Dormand and Prince [19]. Further, a small maximum step size was set to ensure a more uniform triangulation of the surface; this decision increased the integration and surface triangulation time.

5.2 Runtime Environment

All measurements discussed in the following were obtained on the NERSC Cray XT4 system *Franklin*. The algorithms discussed above were implemented in VISIT [1, 6], a visualization system which is routinely used by application scientists. The parallelize-over-data and parallelize-over-particles algorithms were 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. Each benchmark run was performed using 128 cores.

5.3 Measurements

For each process we track the time for I/O, integration, communication, surface triangulation and busy wait. I/O is the total amount of time spent reading from disk. POD only reads the data once where as POP reads data as needed. The integration time should be the same in each group of tests, *i.e.* the narrow short tests for each data set will be the same. Communication time tracks the messages needed for algorithm coordination and the messages needed for the stream surface algorithm to distribute new seed points and

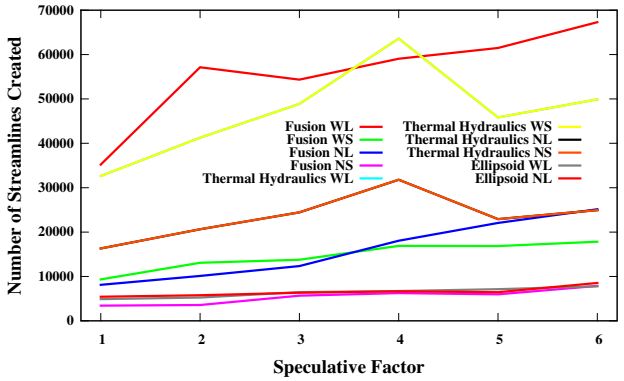

Figure 10: As the speculative factor increases, the number of integral curves created increases. This can lead to more integration time and load imbalance.

distribute integral curves to create the surface. Surface triangulation is the time spent to create the triangle strips for the surface. Busy wait is the time spent by tasks waiting for all other tasks to finish calculating their integral curves. This time shows the load imbalance of the calculations during the refinement step. We also logged the number of integral curves added per refinement step.

6 Results

6.1 Stream Surface

Two timings dominate the overall time spent calculating the stream surface: I/O and busy wait. Communication accounted for a small amount of time in most of the tests. Only in the Fusion wide long tests did the communication time account for a noticeable amount of time. Integration and surface triangulation were a small amount of the overall time. Figure 11 contains a breakdown of time spent for each of the tests.

The speculative refinement factor was used to increase the amount of work done during each refinement cycle. As the speculative refinement factor increased, the number of refinement cycles needed to complete the surface was reduced. This factor did halve the number of refinement steps for most tests with a value of five (see Figure 8). This can be seen in the Fusion wide-long test. With a speculative refinement factor of one, there were nineteen refinement cycles, while with a speculative refinement factor of five, we only needed eight refinement cycles. Of course, the savings from reducing the number of refinement steps comes at the expense of calculating more integral curves. With a speculative refinement factor of five, the average increase in the number of integral curves created was 36%. This increased number of integral curves had different effects on the two integral curve algorithms and we consider them separately in the following sections.

6.2 Parallelize-over-Particles

In most of the POP tests, 40–50% of the time is spent in I/O and 40–50% spent in busy wait. We expected the POP algorithm to be dominated by I/O, but the amount of time spent in busy wait was surprising. The implication is that dividing the initial set of integral curves based on spatial proximity may not be the best assignment pattern. This grouping caused load imbalance on the tasks which contained the longest integral curves as they where placed on the same task. The speculative refinement factor reduced the number of refinement steps needed, which also reduced the overall busy wait time. But, at the same time, it increased the busy wait time per refinement step because the tasks with the longest integral curves had more integral curves to integrate, causing more load imbalance between tasks.

We know that POP performance depends on the number of data blocks that must be loaded to integrate the integral curve. Therefore, POP's performance should improve by processing a large number of integral curves per refinement step to best utilize the data blocks in the cache. The data cache is the heart of this algorithm; we need it to contain the data needed to keep the particle integrating, otherwise, time will be spent reading the new data into memory. But, when the data cache is filled, the POP algorithm removes the least recently used data block when adding a new data block thinking that the oldest data block is least likely needed again. This is problematic; when the stream surface algorithm refines the surface it starts at the seeding curve, meaning the beginning of the data blocks will not be in the cache. So the cache does not help between refinement cycles if the previous integral curves that were created travel though enough blocks to remove some of the beginning data blocks. This case occurred in the Fusion data set. So if we can reduce the number of refinement steps, the cycling of data reads could be reduced.

The speculative refinement factor was used to reduce the number of refinement steps needed to achieve the surface quality, Figure 8 shows the reduction in the number of refinement checks as the speculative factor increases. This reduced number of refinement steps also reduced the number of data blocks loaded; see Figure 9 for the decrease in the number of blocks loaded as the speculative factor increased. We can see the total number of data blocks loaded dropped by 40–50% as the speculative factor increases. So, why don't we see a decrease in the I/O time as the speculative factor increases? Even though the number of data blocks loaded is large, the unique number of data blocks is low, as many of the same data blocks are reloaded. The operating system and I/O system caching appears to reduce the amount of time needed to reload these data blocks. Looking at Figure 11, we see that the time is reduced at higher levels of speculative refinement, but this reduction comes less from a small improvement in I/O time as from a reduction in busy wait time.

6.3 Parallelize-over-Data

Test cases using the POD algorithm highlight the difficulties of load balancing for stream surface calculations. Because the seeding curve is typically contained within a small subset of blocks, only a small subset of tasks will initially be assigned work. This load imbalance is clearly seen in Figure 12, where only 10 of the 128 tasks perform any integration at all, and the majority of integration was done by only two tasks. Additional experiments were performed using different methods of assigning data to tasks (round robin, striping), but the load imbalance remained largely unchanged.

Load imbalance using POD is exacerbated by the need to add refinement seed points. As these new seed points are added along the seeding curve, the same initial small set of tasks will be assigned work, and the rest will remain idle. As the particles are integrated, and move between blocks, the work load may or may not become more balanced, depending on the nature of the vector field. The work imbalance can clearly be seen in the Ellipsoid and Thermal Hydraulics tests in Figure 11. In these two test cases, the dominant factor is busy wait, caused by load imbalance. However, the Fusion test case in Figure 11 exhibits generally good load balance, with less busy wait, and superior performance to the POP algorithm. This is due to the periodic nature of a vector field within a fusion simulation where particles will travel through a much larger percentage of the total data volume. The initial work imbalance along the seeding curve can be overcome as particles circulate within the data volume. Speculative refinement did not prove to be a benefit for most of the POD test cases. It did reduce the number of refinement steps, but at the expense of concentrating a lot of work on a small number of tasks.

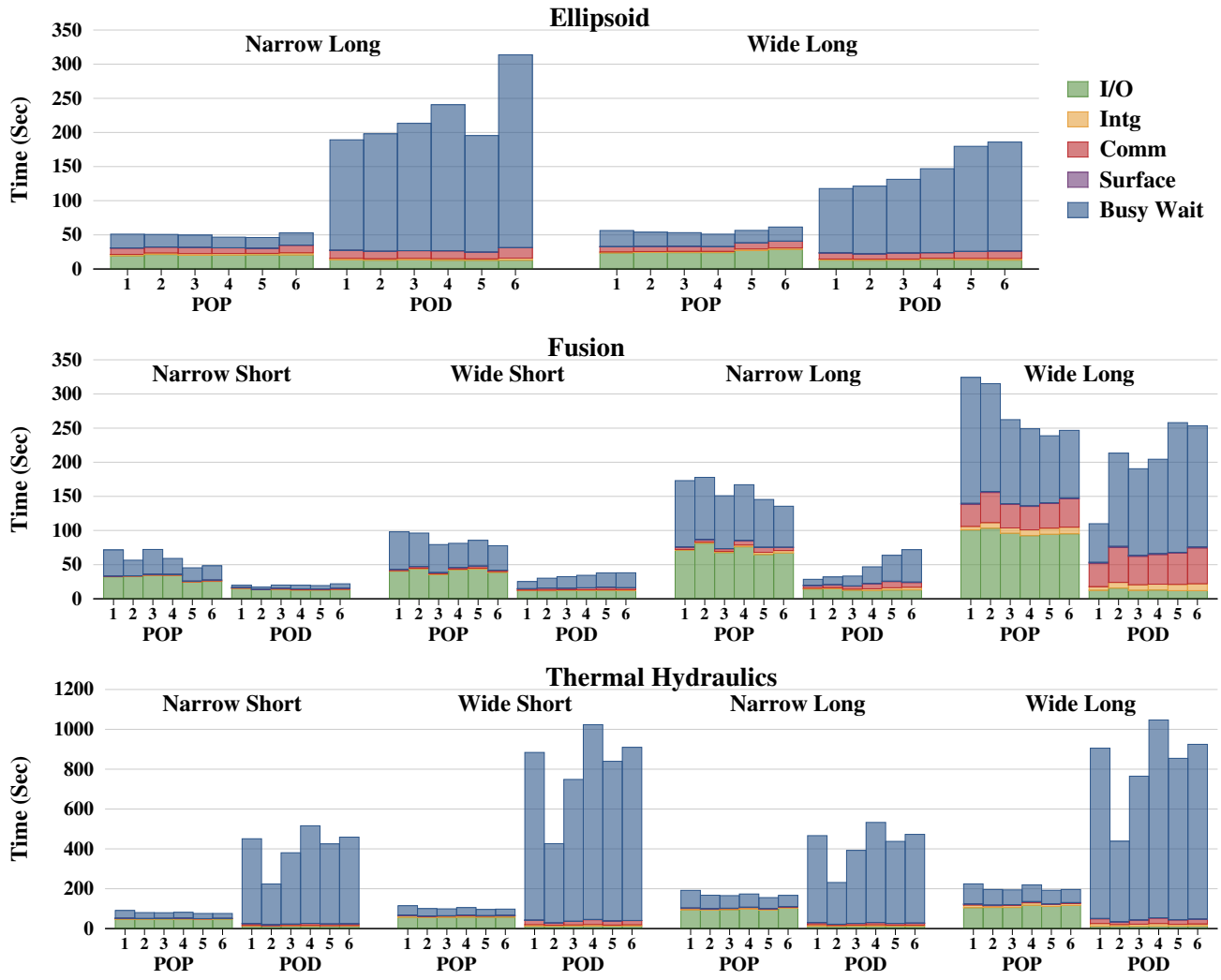

Figure 11: Measurements of our 120 tests. Each column represents one test and are group by the type of test; see Section 5.1 for full description. For each test the following times were tracked: I/O, integration (Intg), communication (Comm), surface generation (Surface) and busy wait times. On the X-axis, the numbers 1 through 6 represent the speculative factor for the test.

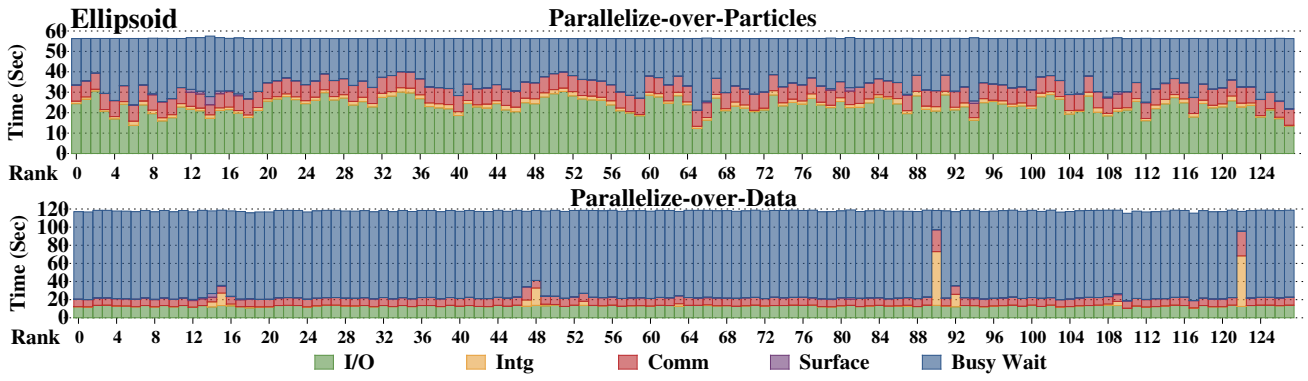

Figure 12: This chart shows the time breakdown for each task during the ellipsoid test. The following items are measured: I/O, integration (Intg), communication (Comm), surface generation, and busy wait times. We can see the load imbalance in the Parallelize-over-Data algorithm, where two tasks are doing much more integration than the others.

7 Conclusion and Future Work

We performed tests to see which parallel particle algorithm is most beneficial to the creation of stream surfaces. We found the POD algorithm performed better in four out of ten tests, although all four of these tests were with the Fusion data set. For the POD algorithm to really perform well, it needs to have a good distribution of integral curves across its tasks and this is not likely with a stream surface. Outside of the Fusion data set, the POP algorithm performed the best, as it was largely unaffected by the integral curve distribution. That said, POP could perform better with changes to the integral curve distributions and data cache retention.

The load imbalance was the biggest factor in all of the tests. Both algorithms spent a lot of time in busy wait. The speculative refinement only helped the POP algorithm. The extra integral curves that were created just added to the load imbalance of the POD algorithm. The POP algorithm was able to reduce the number of data blocks loaded and reduce the busy wait time.

Future work will focus on the weakness of the integral curve algorithms. The POP cache strategy of least recently used seemed to lose a lot of the data blocks. We believe a better caching strategy could improve performance. The POP algorithm could be improved to have better load balancing by a better distribution of the seed points to spread the work out more evenly. The POD could be changed to redistribute the data blocks after the first integral curve calculation to better distribute the computation load. Finally, although this study sought to inform parallel stream surface calculation at the extreme ends of the parallelization spectrum – parallelization over particles or over data – we would like to consider hybrid algorithms, such as the master-slave algorithm [20] and Nouanesengsy's predictive scheduling approach.

8 Acknowledgments

This research used resources of the National Energy Research Scientific Computing Center (NERSC), which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231. This work was also supported by the Director, Office of Advanced Scientific Computing Research, Office of Science, of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231 through the Scientific Discovery through Advanced Computing (SciDAC) program's Visualization and Analytics Center for Enabling Technologies (VACET). The authors wish to thank Markus Rütten from DLR Göttingen for supplying the ellipsoid dataset.

References

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

[2] S. Born, A. Wiebel, J. Friedrich, G. Scheuermann, and D. Bartz. Illustrative Stream Surfaces. *IEEE Transactions on Visualization and Computer Graphics*, 16:1329–1338, 2010.

[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, 2001.

[4] B. Cabral and L. C. Leedom. Highly Parallel Vector Visualization Using Line Integral Convolution. In *Proceedings of SIAM PPSC '95*, pages 802–807, 1995.

[5] L. Chen and I. Fujishiro. Optimizing Parallel Performance of Streamline Visualization for Large Distributed Flow Datasets. In *Proceedings of IEEE VGTC Pacific Visualization Symposium*, pages 87–94, 2008.

[6] 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. In *Proceedings of IEEE Visualization*, 2005.

[7] D. Ellsworth, B. Green, and P. Moran. Interactive Terascale Particle Visualization. In *Proceedings of IEEE Visualization*, pages 353–360, Washington, DC, USA, 2004.

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

[9] C. Garth, H. Krishnan, X. Tricoche, T. Tricoche, 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.

[10] C. Garth, X. Tricoche, T. Salzbrunn, and G. Scheuermann. Surface Techniques for Vortex Visualization. In *Proceedings of Eurographics - IEEE TCVG Symposium on Visualization*, pages 155–165, May 2004.

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

[12] 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*, pages 171–178, Boston, MA, 1992.

[13] M. Hummel, C. Garth, B. Hamann, H. Hagen, and K. I. Joy. IRIS: Illustrative Rendering for Integral Surfaces. *IEEE Transactions on Visualization and Computer Graphics*, 16(6):1319–1328, Nov. 2010.

[14] D. A. Lane. UFAT – A Particle Tracer for Time-Dependent Flow Fields. In *Proceedings of IEEE Conference on Visualization*, pages 257–264, Oct. 1994.

[15] H. Löffelmann, L. Mroz, and E. Gröller. Hierarchical Streamarrows for the Visualization of Dynamical Systems. In W. Lefer and M. Grave, editors, *Proceedings of the 8th Eurographics Workshop on Visualization in Scientific Computing*, pages 203–211, 1997.

[16] 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 *Proceedings of the IEEE Symposium on Parallel and Large-Data Visualization and Graphics (PVG)*, Washington, DC, USA, 2003.

[17] B. Nouanesengsy, T.-Y. Lee, and H.-W. Shen. Load-Balanced Parallel Streamline Generation on Large Scale Vector Fields. *IEEE Transactions on Visualization and Computer Graphics*, 17(12):1785–1794, 2011.

[18] T. Peterka, R. Ross, B. Nouanesengsey, T.-Y. Lee, H.-W. Shen, W. Kendall, and J. Huang. A Study of Parallel Particle Tracing for Steady-State and Time-Varying Flow Fields. In *Proceedings of IPDPS 11*, Anchorage AK, 2011.

[19] P. J. Prince and J. R. Dormand. High order embedded Runge-Kutta formulae. *Journal of Computational and Applied Mathematics*, 7(1):67–75, 1981.

[20] D. Pugmire, H. Childs, C. Garth, S. Ahern, and G. H. Weber. Scalable Computation of Streamlines on Very Large Datasets. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 16:1–16:12, Portland, OR, USA, Nov. 2009.

[21] G. Scheuermann, T. Bobach, H. Hagen, K. Mahrous, B. Hamann, K. I. Joy, and W. Kollmann. A Tetrahedra-Based Stream Surface Algorithm. In *Proceedings of IEEE Visualization*, pages 83–91, 2001.

[22] 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. *Journal of Computational Physics*, 195:355, 2004.

[23] D. Stalling. *Fast Texture-Based Algorithms for Vector Field Visualization*. PhD thesis, Freue Universität Berlin, 1998.

[24] D. Sujudi and R. Haimes. Integration of Particles and Streamlines in a Spatially-Decomposed Computation. In *Proceedings of Parallel Computational Fluid Dynamics*, Los Alamitos, CA, 1996.

[25] 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.

[26] J. J. van Wijk. Rendering Surface–Particles. In *Proceedings of the IEEE Conference on Visualization*, pages 54–61, 1992.

[27] J. J. van Wijk. Implicit Stream Surfaces. In *Proceedings of IEEE Conference on Visualization*, pages 245–252, 1993.

[28] H. Yu, C. Wang, and K.-L. Ma. Parallel Hierarchical Visualization of Large Time-Varying 3D Vector Fields. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '07, pages 24:1–24:12, 2007.