Scalable Computation of Stream Surfaces on Large Scale Vector Fields

Kewei Lu*, Han-Wei Shen* and Tom Peterka†

*Department of Computer Science and Engineering, The Ohio State University, Columbus, OH 43210
Email: {luke,hwshen}@cse.ohio-state.edu

†Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL 60439
Email: tpeterka@mcs.anl.gov

Abstract—**Stream surfaces and streamlines are two popular methods for visualizing three-dimensional flow fields. While several parallel streamline computation algorithms exist, relatively little research has been done to parallelize stream surface generation. This is because load-balanced parallel stream surface computation is nontrivial, due to the strong dependency in computing the positions of the particles forming the stream surface front. In this paper, we present a new algorithm that computes stream surfaces efficiently. In our algorithm, seeding curves are divided into segments, which are then assigned to the processes. Each process is responsible for integrating the segments assigned to it. To ensure a balanced computational workload, work stealing and dynamic refinement of seeding curve segments are employed to improve the overall performance. We demonstrate the effectiveness of our parallel stream surface algorithm using several large scale flow field data sets, and show the performance and scalability on HPC systems.**

I. INTRODUCTION

Effective visualization of flow fields plays an important role in analyzing data generated from scientific simulations, for which displaying streamlines and stream surfaces are two popular methods. A streamline is the trajectory of a massless particle traced from a seed point in the field, and a stream surface is a surface traced from seeds originated form a curve. A stream surface can be seen as the union of an infinite number of streamlines, and is typically approximated by a polygonal mesh that connects streamlines seeded from selected positions on a seeding curve. As the size of data continues to grow, efficient computation of streamlines and stream surfaces becomes increasingly more difficult. To address the challenge, researchers have proposed various parallel computation algorithms, most of which are for streamline computation [1] [2] [3] [4] with a few for stream surfaces [5]. Generally speaking, stream surface computation is more complicated than computing streamlines since stream surface integration requires seeds to be inserted or deleted dynamically across the stream surface front when the flow diverges or converges. This dynamic insertion and deletion of seeds requires synchronization among the computation of individual streamlines, and thus makes parallel computation of stream surfaces much more challenging.

Algorithms for parallel streamline computation can be divided into two types: *parallelizing-over-seeds* and *parallelizing-over-data* methods. For the parallelizing-over-seeds methods, seeds are distributed across all processes, and then each process computes the streamlines originated from the assigned seeds and loads the required data when necessary. For the parallelizing-over-data methods, data are first decomposed into blocks, and these data blocks are distributed to the processes. Each process computes the streamlines in its own blocks and sends the streamlines to the other processes once the streamlines hit the block boundaries. Applying these two strategies directly to parallel stream surface computation is nontrivial because of the dependency in the adjacent streamlines that form a stream surface. When using the parallelizing-over-seeds strategy, for example, if two adjacent seeds are distributed to different processes, a considerable amount of communication is needed to connect the streamlines to a surface. On the other hand, if parallelizing-over-data is used, challenges arise when the same seeding curve passes multiple data blocks that belong to different processes.

In this paper, we present a scalable parallel stream surface computation algorithm based on a front-advancing approach that is also used by several sequential stream surface algorithms [6] [7] [8]. The main contribution of this paper is an efficient algorithm to compute stream surfaces for large scale vector fields that can handle flow convergence, divergence and split. We show that our algorithm is highly scalable when running on large supercomputers. We use a hybrid parallelizing-over-data and parallelizing-over-seed strategy to ensure a balanced workload distribution for machines with large processor counts. In our algorithm, we divide the seeding curve into segments and then assign these segments to different processes so that each process computes a part of the surface. For each seeding segment, the seeds at the boundary are duplicated so that each seeding segment can be advanced independently by the processes. We divide the entire data set into blocks which are distributed across the processes. During integration, when a process needs a data block not in its local memory, the process will get the block from the other process over the network on demand. Load balancing is ensured by a work stealing mechanism combined with a runtime seeding curve partitioning scheme.

The rest of the paper is organized as follows. In Section II, we review the related works in parallel streamline and stream surface algorithms. In Section III, we describes our algorithm in detail, including preprocessing, parallel computation, and load balancing. Section IV presents several experimental results to study the scalability of our method. Conclusion and future work are presented in Section V.

II. Background

Since steam surfaces are closely related to streamlines, in this section we first briefly review parallel streamline generation techniques and the common parallel strategies that are used. Then we discuss the existing algorithms for stream surface generation and the previous work for parallel stream surface computation.

A. Parallel Streamline Computation

A streamline is defined as a curve traced from a seed location in the flow field and is tangential everywhere to the local flow direction. It is computed by applying numerical integration techniques such as the Runge-Kutta methods to obtain a sequence of positions starting from a user-specified seed location. To parallelize streamline computation, two strategies are often used. One is to decompose the entire data set into a number of disjoint blocks and then distribute those blocks to the processes. With the block assignment, each process will compute the streamline segments only if they pass through their own blocks. This parallelizing-over-data strategy was used in an early parallel streamline computation method proposed by Sujudi and Haimes [9]. Based on this idea, Peterka et al. [4] presented a study of parallel particle tracing for both streamline and pathline generation for steady and unsteady flow fields. Kendall et al. [10] proposed a MapReduce [11] like system called $DStep$ for parallel streamline computation. Nouanesengsy et al. [3] addressed the load balancing issue in the parallelizing-over-data approach and proposed a workload-aware partitioning algorithm based on a graph representation of the original flow field. The second strategy of parallel streamline computation is parallelizing-over-seeds, where each process computes streamlines from the seeds that are assigned to it and loads the data blocks required to complete the integration of the seeds on demand. Pugmire et al. [2] reviewed both strategies and presented a new hybrid approach for streamline computation.

B. Parallel Stream Surface Computation

A stream surface is defined as a surface traced from a seeding curve inside the flow field. An ideal stream surface can be constructed by densely sampling an infinite number of seeds on the seeding curve and connecting the resulting streamlines together. The streamlines originated from the seeding curve can be parametrized by their arc length $t \in [0, N]$. For a constant t, the positions of all streamlines form a front, sometimes referred to as a time line, and can be parameterized by another parameter $s \in [0, 1]$, which indicates the originating position of the streamline on the seeding curve. These two sets of curves, streamlines and the fronts, define the stream surface parametrized over s and t. Any point on a stream surface can be represented as:

$$\Upsilon(s,t) = X_{C(s)}(t) \qquad (1)$$

where C is the seeding curve parametrized by s, $X_{C(s)}(t)$ defines a curve traced from a seed $C(s)$ on C and parametrized by t; and Υ defines the stream surface constructed by advancing C and is parametrized by s and t. Figure 1 illustrates the parameterization of a stream surface.

In practice, tracing an infinite number of streamlines to construct the stream surface is not feasible. A front-advancing
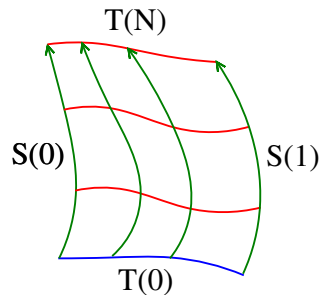

Fig. 1: A stream surface parametrized by s and t. The blue curve is the initial seeding curve. The red curves, also called time lines, can be seen as the front of the seeding curve moved by the flow direction.

algorithm for generating a stream surface was first presented by Hultquist [6]. In the algorithm, the seeding curve is discretized by a number of seed points, which are then advanced by integrating these seed points with a numerical method. Adjacent pairs of streamlines form stream ribbons which are triangulated by a greedy method. Adaptive refinement such as insertion and deletion of streamline seeds is employed in Hultquist's algorithm to control the resolution of the advancing front. Splitting of the surface is handled by comparing the advancing direction of adjacent seeds in Hultquist's algorithm. While Hultguist's algorithm is simple to implement, it may not perform well when the flow in local regions has large variations in direction and magnitude. Later, an improved algorithm was proposed by Garth et al. [7]. In their algorithm, the authors employ an arc length based streamline integration scheme. Also, additional front refinement criteria such as surface curvatures were introduced.

In addition to Hultquist's and Garth's algorithms that construct stream surfaces explicitly, there exist methods that construct stream surfaces implicitly [12] [13]. In the method by van Wijk [12], continuous scalar values are specified for the grid points on the boundaries of the flow field, and then the scalar values for the interior grid points are computed by backward tracing of streamlines. With the scalar field, stream surfaces can be constructed by extracting isosurfaces. Stöter et al. [13] extended van Wijk's approach to stream, path, streak, and time surfaces for 3D time-varying flow fields and solve some limitations of van Wijk's approach such as limited domain coverage and limited control of the seeding curve.

Compared with streamlines, parallelization of stream surfaces is more challenging because of the dependency in computing the streamline points that form the stream surface front. This dependency can be seen from the sequential stream surface algorithms [6] [7] [8], which construct stream surface by advancing the stream surface front step by step across the seeding curve and applying adaptive refinement at every step to ensure the accuracy of the stream surface. This dependency makes both the parallelizing-over-data and parallelizing-over-seed strategies inefficient because excessive communication between processes may occur. Previously, Camp et al. presented a parallel stream surface algorithm [5] that does not utilize the front-advancing method. Instead, in their method the stream surface is approximated by tracing streamlines from
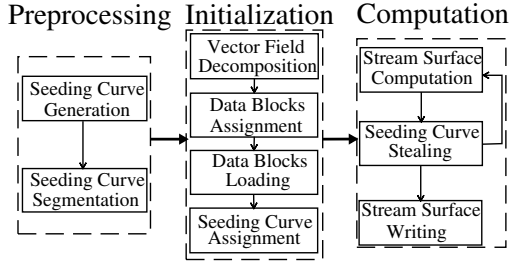
Fig. 2: An overview of the pipeline used in our algorithm. The preprocessing stage runs in serial, and the initialization and computation stages run in parallel.
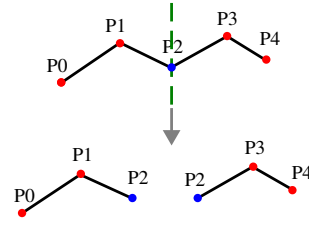

Fig. 3: An example of our seeding curve segmentation. Given a seeding curve with five particles P_0 to P_4 shown at the top, we uniformly divide it into two pieces at particle P_2. The particle P_2 is the boundary particle after the cutting, and is duplicated for both of the two seeding curve segments.

seed points on the seeding curve and triangulation is performed afterwards. Surface refinement is achieved by inserting new seeds between two adjacent seed points if the distance between the streamlines originating from these two seeds is larger than a predefined threshold. This refinement process stops when no adjacent pair of streamlines exceeds the threshold. Since it is likely that the discontinuities in the flow field may prevent the refinement of a stream surface from stopping, the algorithm does not insert new seeds if the distance between the adjacent seed points is less than a threshold.

In this paper, we present a parallel algorithm utilizing the front-advancing scheme for explicit stream surface computation. We present a hybrid parallelizing-over-data and parallelizing-over-seed algorithm to keep the I/O cost low and to ensure a balanced workload distribution. With the front-advancing scheme, our algorithm can address some of the limitations of Camp's algorithm such as no seed deletion is done when flow begins to converge, resulting too many unnecessary points being generated on the stream surface. Also, in Camp's algorithm, seed insertion only takes place on the original seeding curve, and hence no adaptive refinement is performed across the surface. Below we describe our algorithm in detail.

III. METHOD

In this section, we present our parallel algorithm for stream surface computation based on a front-advancing method. Our algorithm is closely related to Garth's algorithm [7], but other front-advancing methods [6] [8] can also benefit from our parallelization strategy. In our implementation, we are using OSUFlow, a parallel particle advection library [4] developed by The Ohio State University and Argonne National Laboratory. In our implementation, the Runge-Kutta-Crash-Karp($RK45$) numerical integration scheme is used. DIY [14] is used to decompose the domain, assign data blocks, and perform inter-process communication. The Block I/O layer (BIL) library is used to achieve high I/O efficiency when loading disjoint data blocks across the processes [15]. By using BIL, each process posts requests for the required data blocks, and then a single collective I/O is issued to read the data in parallel. MPI-I/O is used to write the final results to disk in parallel.

An overview of the pipeline in our algorithm is shown in Figure 2, which can be divided into three main stages: preprocessing, initialization and stream surface computation.

Algorithm 1 Seeding Curve Generation

1: Let n be the number of seeding curves, m be the maximum number of seeds per seeding curve, and s be the distance between adjacent seeds on the seeding curves
2: **for** $i = 1$ to n **do**
3:    Randomly generate two seeds p_0 and p_1
4:    $dir \leftarrow$ normalized vector points from p_0 to p_1
5:    $d \leftarrow$ distance between p_0 and p_1
6:    Initialize l to be an empty list
7:    Initialize $curDistance \leftarrow 0$
8:    **for** $j = 0$ to m **do**
9:       $p \leftarrow p_0 + j \times s \times dir$
10:      $curDistance \leftarrow curDistance + s$
11:      Push p to l
12:      **if** $curDistance > d$ **then**
13:         $Break$
14:      **end if**
15:    **end for**
16:    Add l to the seeding curve pool
17: **end for**

The preprocessing stage does not need run in parallel since the computation cost of this stage is negligible compared with the initialization and computation stages. In our implementation, the preprocessing stage runs in serial, and the initialization and computation stages run in parallel.

A. Preprocessing

The preprocessing stage includes seeding curve generation and seeding curve segmentation. Seeding curves can be either provided by the user or randomly generated. In our implementation, we randomly generate the seeding curves, where the user specifies how many seeding curves to generate, the maximum number of seeds on each seeding curve, and the distance between adjacent seeds on the seeding curve. Generating seeding curves is shown in Algorithm 1.

Given a set of seeding curves, we partition them into segments that have an equal number of seeds to balance the computational workload. The seeds at the boundaries of the seeding curve segments, referred to as the boundary seeds, are duplicated so that they are in both the left and the right segments as shown in Figure 3. The duplication of the boundary seeds eliminates the need to communicate when

advancing the surface fronts between different processes.

B. Initialization

The purpose of initialization in our algorithm is to distribute the data and the seeding curve segments to the processes. In this stage, first the vector field is decomposed into axis-aligned 3D blocks, where the number of blocks should be greater or equal to the number of processes. To allow a continuous interpolation of data in each block independently, one layer of ghost cell is added to the boundary of each dimension. After the data decomposition, different blocks assignment strategies can be used such as the round-robin and the processes-order continuous schemes [4]. The round-robin assignment scheme tends to perform better than the processes-order continuous assignment scheme for parallel streamline computation as illustrated by Peterka in [4] because a better computation load balancing can be achieved. However, in our parallel stream surface algorithm since we use runtime seeding curve segment partitioning and work stealing to solve the load balancing problem, the choice of data assignment scheme is not as important. In our implementation, we use a simple round-robin data assignment scheme, in which data blocks are assigned to processes by using a block-cyclic distribution.

Besides the data blocks, seeding curve segments also need to be distributed to the processes. Similarly, we can employ various strategies to assign the seeding curve segments. In our algorithm, we assign seeding curve segments to processes in round-robin order.

C. Parallel Stream Surface Computation

The next stage in our algorithm is to compute the stream surfaces by parallelizing the computation of stream surface patches originated from the seeding curve segments. Then, the individual stream surface patches computed by different processes are combined together to form a complete stream surface. Our parallel algorithm employs Garth's front-advancing algorithm to complete each stream surface patch, with several strategies to ensure a scalable parallel performance. We discuss these strategies below.

1) Data Loading On Demand: Our algorithm is parallelized over seeding curve segments. Seeding curve segments are first distributed across the processes evenly, and then the processes integrate the assigned stream surface patches in parallel until the maximum number of steps for each patch is reached or the patch goes out of bounds. During integration, data blocks are loaded into memory when necessary. Unlike the load on demand implementation presented in [2] where data blocks are loaded from disk and I/O can become a bottleneck, we load all the data blocks from disk in the initialization stage as described in section III-B, and then during integration, if the required block is not available locally it will be requested from the process who owns it via communication. In our algorithm, a new data block is needed when the particle (the current position of a streamline) on the stream surface front requires it for the integration. Because all particles on the stream surface front need to be integrated one step together to test the flow divergence or convergence condition, when a particle needs a new block, we need to immediately obtain it to continue the surface patch computation. When no more local memory is
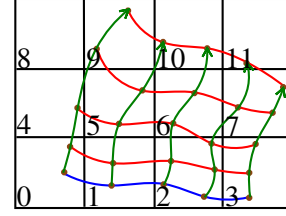

Fig. 4: An example that shows the data access pattern of stream surface integration. The blue curve is the seeding curve segment with five particles. Since the front of the seeding curve segment is advanced one step at a time by integrating the particles. Data blocks are repeatedly accessed.

available to accommodate the new block, one of the in-core blocks needs to be evicted to make room for the block.

In our algorithm, each process maintains a list of the assigned seeding curve segments $\{c_0, c_1, ..., c_n\}$ and computes the stream surface patches one at a time. During the computation, particles on the seeding curve segment are integrated one step further to advance the stream surface front. When process P_i integrates a particle, it first checks which data block this particle needs, and then checks whether this data block is already in memory. The block is available in the local memory either because this block was loaded in the initialization stage, or it has already been requested from another process. If the block is not available locally, first P_i checks which process has the data block and then sends a message to that process say P_j asking for the data block. When process P_j receives the request, it sends the data block to P_i, and in the mean time still keeps the original copy of the data block.

2) Cache: To minimize the number of times it requests data blocks from the other processes, each process maintains a small cache to store the data blocks after they are received. The Least Recently Used (LRU) replacement policy is employed. As will be shown in section IV, the cache helps reduce the number of data requests a process issues to the other processes; this is because our method does not access data blocks randomly but has a specific data access pattern which is favored by the cache. Given a seeding curve segment, the front of the segment is advanced by iteratively integrating all the particles on the seeding curve segment together, which means each data block is accessed repeatedly during the computation of a stream surface patch in a short amount of time. A 2D example is shown in Figure 4. The 2D vector field data is decomposed to 12 blocks with ID from 0 to 11. The blue curve is the initial seeding curve segment with five particles on it, and it is advanced for four steps. Suppose we integrate from left to right, blocks $\{0, 1, 2, 2, 3\}$ are accessed in the first iteration. In the next iteration, we still access blocks $\{0, 1, 2, 2, 3\}$, the same blocks as before. Considering all these four steps, the order of data blocks to be accessed is: $\{0, 1, 2, 2, 3, 0, 1, 2, 2, 3, 4, 5, 6, 2, 3, 9, 5, 6, 7, 7, 9, 10, 10, 11, 7\}$. Notice that data blocks are repeatedly accessed frequently which makes a cache beneficial.

3) Dynamic Load Balancing: In this section, we discuss our dynamic load balancing strategies, namely work stealing

Algorithm 2 Main Structure

1: Partition domain and tasks
2: **for** all data blocks assigned to my process **do**
3:     read the data blocks
4: **end for**
5: **for** all tasks assigned to my process **do**
6:     read the tasks to my task pool T_p
7: **end for**
8: **while** My task pool T_p is not empty **do**
9:     Get a task from the head of T_p and execute
10: **end while**
11: Steal tasks from other processes until no task can be stolen

---

Algorithm 3 Work Stealing

1: **while** *my task pool is empty* and *no global termination* **do**
2:     Randomly select a victim p_v
3:     Send a message to p_v to ask for tasks and wait for a reply
4:     **if** p_v has tasks available **then**
5:         Get half of tasks t from the tail of p_v's task pool
6:         Put t to my task pool
7:     **end if**
8: **end while**

Fig. 5: An example that shows the structure of a seeding curve segment. The yellow header includes three floating point numbers representing the number of steps this seeding curve segment has advanced, the ID of the original seeding curve it comes from, and the number of particles on the seeding curve. The red part is the particles. The green part is the current step size for each particle and the blue part is the number of steps that have been integrated for each particle.

and runtime seeding curve segment subdivision. A high level description of our algorithm is listed in Algorithm 2.

As mentioned in section III-C1, each process maintains a list, referred to as the task pool T_p, to store the assigned seeding curve segments $\{c_0, c_1, ..., c_n\}$, each of which in the pool is a task. To improve load balancing of our algorithm, a dynamic load balancing scheme based on work stealing [16] is employed. We call the process who steals tasks a *thief* and the process whose tasks are stolen a *victim*. When a process finishes all its tasks, it steals tasks from the other processes. To do this, the thief must first select a victim that has extra unfinished tasks. This selection of victim is done based on a random selection method that has been proven optimal [17]. Once a victim is selected, an MPI message is sent to the victim to ask for tasks. When the victim receives the message, it checks its task queue to see whether there are extra tasks. If there are available tasks, half of the tasks starting from the end of the victim's task queue are sent to the thief. Previously researchers have shown that stealing half of the available tasks each time can generate satisfactory results. Since tasks are distributed to more processes, it is easier for the idle processes to find tasks to steal [16]. On the other hand, If the victim has no tasks available, it sends a message back to inform the thief. The thief will then select a new victim randomly and repeats the process until either it finds available tasks or a global termination of the program is detected. Work stealing is shown in Algorithm 3.

Runtime seeding curve segment subdivision means that during integration, if a seeding curve segment grows too wide as the result of flow expansion, the segment is split and half of the segment is pushed to the end of the task pool T_p. The purpose of seeding curve segment subdivision is to split the seeding curve segments that have high workload into smaller pieces, which allows better workload distribution when combined with our work stealing scheme. In our experiments, we found that the number of particles on a seeding curve segment provides a good approximation of the workload. To incorporate this idea, in our implementation we check the number of particles on the seeding curve segments after every integration step. If it is greater than a user defined *cut threshold* Th_{cut}, the middle particle on the segment is selected as the partitioning particle. As in the segmentation of the seeding curves in the preprocessing stage, the partitioning particle is duplicated to the two resulting seeding curve segments. We continue advancing one of the two new seeding curve segments and put the other one to the end of the task pool. Runtime

subdivision is shown in Algorithm 4.

4) Global Termination Detection: To issue a global termination of the parallel program, we have to detect when all the processes are idle and have no more work to do. In our algorithm, we adopted Francez's algorithm [18] for this purpose. We organize the processes into a binary tree, and the termination process involves sending messages up and down the tree in multiple rounds. In each round, messages are propagated from bottom up first. When the 'root' receives messages from its children, it determines whether the program should be terminated or not and propagates the decision up and down. This multi-round communication continues until a global termination condition is detected, which means all processes become thieves and no process is a victim since the last round.

5) Data Structure: Since the original seeding curves are partitioned into multiple segments and these segments may further be subdivided later, when a seeding curve segment is stolen, additional information such as the original seeding curve it belongs to needs to be sent to the thief with the seeding curve segment. In our implementation, the data stored in the data structure of a seeding curve segment include four parts as shown in Figure 5. The yellow header in Figure 5 contains three floating point numbers that record the number of steps this seeding curve segment has advanced, the ID of the original seeding curve it comes from, and the number of particles on the seeding curve. For n particles on the seeding curve, the next $n \times 3$ floating point numbers shown in red color record the 3D positions of the particles on the seeding curve segment. The green part contains n floating point numbers recording the current step size for each particle on the seeding curve segment. Because we use the RK45 integration scheme with adaptive step size, the step size of each particle is dynamically changed based on the flow complexity. If a seeding curve is stolen by another process, this information
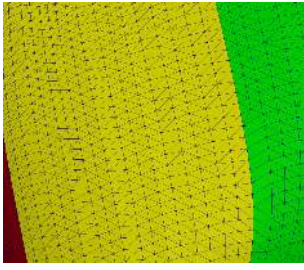
Fig. 6: By sending the current step size of each particle along with the seeding curve segment to the thief, the boundaries of different patches match with each other.

Fig. 7: An example that shows the structure of the message containing the tasks stolen by the thief. Suppose the thief stole N seeding curve segments. The yellow part is the header that includes $N + 1$ floating point numbers. The first number represents the number of the seeding curve segments. The following N numbers are the length of each seeding curve segment. The remainder is the N seeding curve segments.

is needed for the thief to continue to integrate the particles with the right step size so that the boundary of each patch matches as shown in Figure 6. The last n floating point numbers store the number of steps that each particle has integrated. In either Hultquist's or Garth's algorithm, as the stream surface front moves forward, whether a particle should advance or not for this iteration depends on whether the flow is divergent, convergent or neither. This makes it necessary to record the number of steps that each particle has integrated since different particles may have traveled a different distance. Given this seeding curve segment data structure, each seeding curve segment is a $5 \times n + 3$ floating point array.

As explained above, the work stealing victim sends half of its seeding curve segments from its task queue to the thief. These seeding curve segments are organized as a single message. The structure of this message is shown in Figure 7. Suppose the message contains N seeding curve segments, the header shown in yellow color contains $N + 1$ floating point numbers. The first number records how many seeding curve segments are contained in this message. The next N numbers record the length of each segment. The rest are the seeding curve segments. So each stealing operation results in a length of $N + 1 + \sum_{i=0}^{N}(5 \times N_i + 3)$ floating point numbers to be communicated between processes, where N_i is the number of seeds on the ith seeding curve segment.

IV. RESULTS

In this section, we present the performance of our parallel stream surface algorithm. We conducted several experiments using four steady flow field datasets of different resolutions. The dataset *Isabel* models a strong hurricane in the West Atlantic region in September 2003. Its size is 287MB with a resolution of $500 \times 500 \times 100$. We randomly generated 256 seeding curves, each with a maximum 128 seeds and the

---

Algorithm 4 Runtime Subdivision
___
1: Given a seeding curve segment in the task pool T_p
2: **while** The maximum number of steps is not reached and the seeding curve segment has not hit the global boundary **do**
3:     Advance one step further
4:     **if** The number of particles on the current seeding curve segment $> Th_{cut}$ **then**
5:         Divide the seeding curve segment at the middle seed resulting in two new seeding curve segments C_1 and C_2
6:         Set the current seeding curve segment to C_1
7:         Put C_2 to the tail of the task pool
8:     **end if**
9: **end while**
___

distance between the seeds was 0.5 cell. *Madden-Julian Oscillation(MJO)* is a dataset simulating the Madden-Julian oscillation effect over the Indian and Pacific Oceans. It is 926MB with a resolution of $2699 \times 599 \times 50$. 256 seeding curves were randomly generated, each with a maximum of 128 seeds and 0.5 cell between the seeds. The *Plume* data set is generated from a simulation of solar plume on the surface of the sun with a resolution of $504 \times 504 \times 2048$. The data size is 5.9GB. 64 seeding curves were generated, each with a maximum of 512 seeds and 0.5 cell sample distance between the seeds. The last test dataset, referred to as *Nek*, is a simulation of thermal hydraulics generated by the Nek5000 solver. It was created from a large-eddy simulation of the Navier-Stokes equation for the MAX experiment [19]. It has a resolution of $2048 \times 2048 \times 2048$ for a total size of 96GB. 64 seeding curves were generated, each with a maximum of 2048 seeds and 0.5 cell between the seeds. Figure 8 shows images of the stream surfaces computed from these four datasets.

For all the experiments, we measured the time spent on stream surface computation and communication for each process. The computation time, referred to as Comp, is the total amount of time spent on advancing seeding curve segments using the Runge-Kutta-Crash-Karp integration scheme. The reported computation times are the average computation time per process. The communication time, referred to as Comm, measures the time for sending and receiving messages to get data blocks from the other processes, work stealing, global termination detection, and the time to manage communication. The average communication time per process is reported.

A. *Parameter Setting*

There are several parameters which influence the performance of our parallel stream surface computation algorithm: the block size, the cache size, the threshold used to divide the seeding curves in the preprocessing stage (referred to as the preprocessing threshold), and the threshold used to divide the seeding curves at run time (referred to as the runtime threshold). We conducted experiments to study the influence of these parameters to the performance of our algorithm. All tests were conducted on the Blue Gene/Q *Vesta* system at the Argonne Leadership Computing Facility. *Vesta* has 2,048 nodes, each holding 16 PowerPc A2 1600MHz cores sharing 16 GB of RAM and utilizes the General Parallel File System.

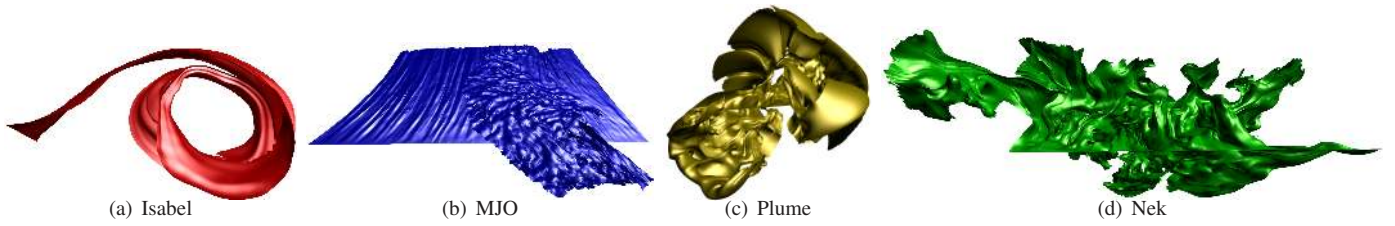(a) Isabel   (b) MJO   (c) Plume   (d) Nek

Fig. 8: Images of a single stream surface computed from the datasets Isabel, MJO, Plume, and Nek respectively.
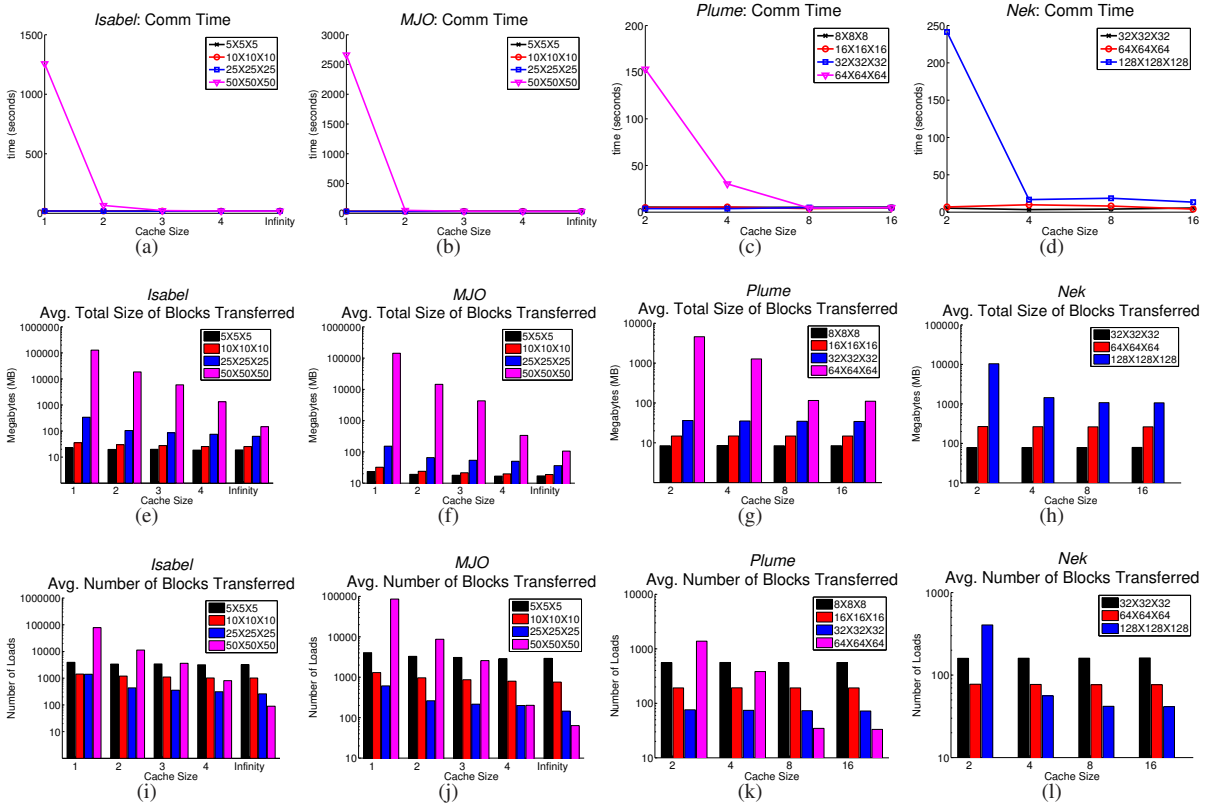

Fig. 9: Top row: The cache size versus communication time under different block sizes. Middle row: The cache size versus the average total size of data blocks transferred per process under different block sizes. Bottom row: The cache size versus the average number of block loads per process under different block sizes.

The total memory is 32 TB.

1) Block size Versus Cache Size: Among the four parameters mentioned above, the block size and the cache size influence the time to load data blocks from the other processes the most. To study the impact of these two parameters, we performed a sequence of experiments using all four datasets. For *Isabel* and *MJO*, the block sizes tested were 5^3, 10^3, 25^3, and 50^3. Blocks of 8^3, 16^3, 32^3, and 64^3 were tested for *Plume*, and blocks of 32^3, 64^3, 128^3 were tested for *Nek*. These sizes do not include the ghost layer, although the actual data block loaded has one layer of ghost cells in each dimension. In our tests, 128, 512, 1024, and 1024 processes were used for *Isabel*, *MJO*, *Plume*, and *Nek* respectively. The size of the cache is controlled by the number of blocks N that it can hold. For the *Isabel* data set, for example, a cache that can hold N blocks means the amount of memory allocated to the cache is N

$\times$ 50^3 vectors. Therefore, with the same amount of memory, more than N blocks that have a smaller size such as 5^3, 10^3 and 25^3 can fit to the cache. In our experiments, the cache size was varied with values of $1, 2, 3, 4$, and infinity for *Isabel* and *MJO*, where infinity means the amount of memory allocated to the cache is able to hold the entire dataset. For *Plume* and *Nek*, the cache size was varied with values of $2, 4, 8$, and 16. As for the stream surfaces generated in our test, the maximum number of integration steps was set to $400, 200, 600$, and 500 for *Isabel*, *MJO*, *Plume*, and *Nek*, respectively. The seeding curve partitioning thresholds, i.e., the width of the seeding curve segment in the preprocessing stage, and the threshold for the run time seeding curve partitioning were fixed across all experiments. The influences of these two parameters on our algorithm will be studied in section IV-A2.

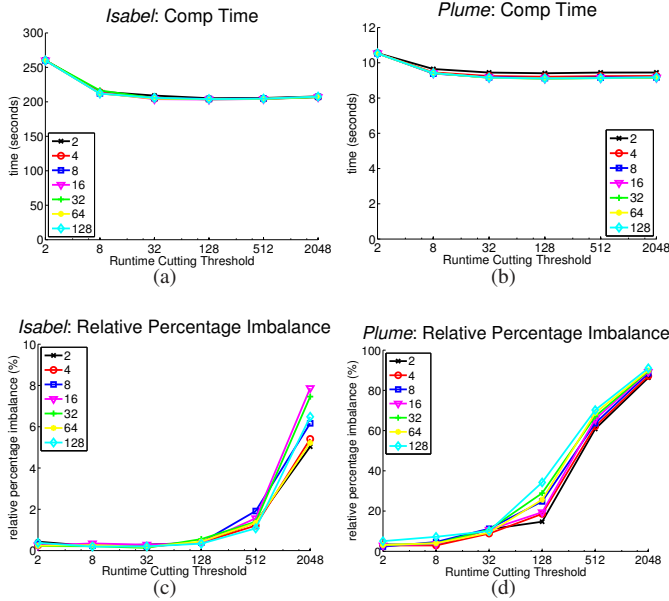Figure 9 shows the results of our tests. Besides the com-

Fig. 10: Top row: Runtime cutting threshold versus computation time under different static cutting thresholds. Bottom row: Runtime cutting threshold versus relative percentage imbalance under different static cutting thresholds.

munication time, we also measured and reported the average amount of data transferred by each process, and the average number of data blocks loaded by each process. Previously Peterka et al. [4] reported that for parallel streamline computation, using small block size and round-robin block assignment will distribute workload more evenly, but the downside is that it incurs more communication. In our tests, we found that smaller block sizes did not improve load balancing since the workload of a process mainly depends on the seeding curve segments instead of the block size. In general, a smaller block size results in less data to be transferred among processes because smaller blocks pack the stream surface front more tightly and hence less unnecessary data are transferred. This is shown in Figure 9(e)-9(h).

As the cache size increases, the number of blocks that were loaded decreases. When the cache size is large enough or goes to infinity, using a larger block size requires fewer loads compared with a smaller block size, because a smaller block size implies the stream surface front will pass through more blocks. Figure 9(i)-9(l) show this trend. The influence of block size for the communication cost is shown in Figure 9(a)-9(d). For a fixed cache size, using a large block size will increase the data loading time, if the cache is not large enough to load the required blocks to cover the stream surface front. As cache size increases, the difference between different block sizes vanishes.

2) Seeding Curve Cutting Thresholds: The seeding curve cutting thresholds in the preprocessing stage and at run time also can influence performance and load balance. Hereafter we refer to the threshold used in the preprocessing as *static cutting threshold*, and the threshold used at run time as *runtime cutting threshold*. We conducted tests using *Isabel* and *Plume*

with 128 and 1024 processes to study the impact of these two thresholds. The block size was set at 25^3 for *Isabel* and 32^3 for *Plume*. For both tests, we used a large cache size to minimize the number of data loads so that we can focus on the computation time and workload balance. Cache size for *Isabel* was 1600 and for *Plume* was 200. Other parameters used are identical to those in section IV-A1.

The results of our test are shown in Figure 10, where curves of different colors represent the computation times using different static cutting thresholds, and the x axis represents the runtime cutting threshold. As can be seen in Figure 10(a) and 10(b), as we increase the value of the runtime cutting threshold, i.e., making the seeding curve segments wider, the decreases in the computation time are noticeable at the beginning but then stop later. This is because when the runtime cutting threshold is too small, such as 2, too many seeds are duplicated, hence increasing the computation time. On the other hand, when the runtime cutting threshold increases, the number of duplicated seeds decreases, decreasing the computation time. When the runtime cutting threshold becomes even larger, such as 32, the duplicated seeds are only a small fraction of the total number of seeds, so the decrease in computation time becomes negligible.

We use the formula $\left(\frac{t_{max} - t_{avg}}{t_{max}}\right) \times 100\%$, referred to as the relative percentage imbalance, to measure the load balancing of our algorithm, where t_{max} is the maximum computation time taken by a process and t_{avg} is the average computation time over all processes. The relative percentage imbalance are shown in Figure 10(c) and 10(d). The increase of relative percentage imbalance when the runtime cutting threshold becomes larger indicates that workloads among the processes become imbalanced. The reason for this is that when using a larger threshold, fewer seeding curve segments are produced so that it is harder for a process to find work to steal. Also, larger thresholds produce longer seeding curve segments with larger differences between the workload of each. From the figures, we can also see that the static cutting threshold does not have a significant impact on the performance of our algorithm compared with the runtime cutting threshold. However, we note that the static cutting threshold should not be too small, such as 2. This is because if the flow converges, different seeding curve segments can not be merged at runtime. Generally speaking, extremely small or large cutting thresholds degrade the performance. Considering the tradeoff between having more work and being less load balanced, in our experiments, a value between 32 and 128 was best.

3) Cache Size Versus Runtime Cutting Threshold: In this section we investigate the relationship between the runtime cutting threshold and the cache size. We conducted tests using *MJO* and *Nek* with 512 and 1024 processes, respectively. The block size was 25^3 for *MJO* and 32^3 for *Nek*. Different runtime cutting thresholds were tested for each dataset. The cache size was represented by the maximum number of blocks that can be stored and varied for each runtime cutting threshold. Other parameters are identical to those in section IV-A1.

The results in Figure 11(a) and 11(b) show that as the cache size increases, the communication time decreases at the beginning and then becomes stable. When a smaller cache size is used, fewer number of blocks could be loaded which are
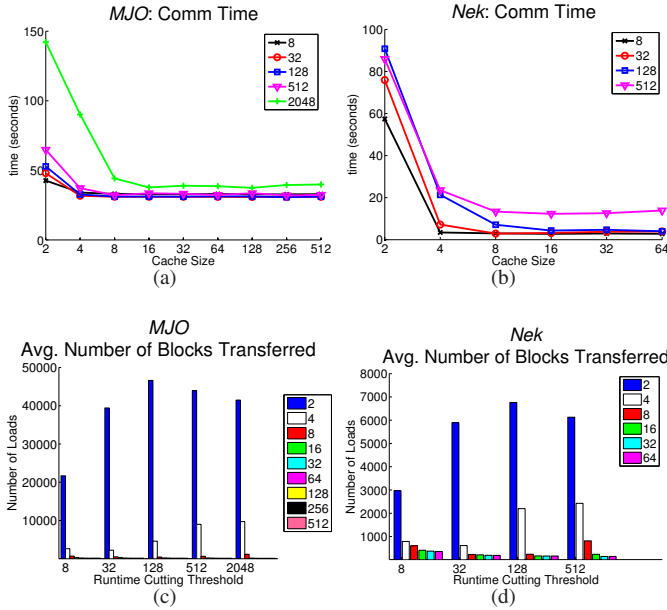
Fig. 11: Top row: Cache size versus communication time under different runtime cutting thresholds. Bottom row: Runtime cutting threshold versus average number of block loads under different cache sizes.

not enough to cover the entire stream surface front, resulting in too many data blocks to be loaded on the fly. Increasing the cache size allows more data blocks are loaded, resulting a decreased number of data block loads. As we continue to increase the cache size, at some point the performance is not improved. This is because the current cache size is large enough to load sufficient blocks to cover the surface front. This point depends on the runtime cutting threshold because a larger runtime cutting threshold produces longer seeding curve segments which generally need more data blocks.

B. Discussion

Among these four parameters, the runtime cutting threshold determines the total amount of particle tracing to be done and the load balancing of our algorithm. Our experiments suggest that the runtime cutting threshold should not be too small; otherwise, it will increase the total amount of work due to seed duplication. On the other hand, if the threshold is too large, load imbalance occurs. In our experiments, a value between 32 and 128 was best. The static cutting threshold should be smaller than the runtime cutting threshold to avoid immediately partitioning at runtime but not extremely small such as 2. The cache improves the performance of our algorithm by holding the recently used data blocks and reducing the number of data blocks loads. The cache size is related to the block size. Given a fixed amount of memory allocated to the cache, more blocks of smaller sizes could be loaded compared with blocks of large sizes. Given a fixed block size, the ideal cache size is to be able to load the blocks that cover the stream surface front, which is related to the runtime cutting threshold. In general, a smaller block size is preferred since it incurs less data transfer among the processes and also they can cover the stream surface

front more tightly. However, a side effect of using a smaller block size is that it will cause more data loading operations. When an extremely small block size is used, a larger number of data loading will take place, and hence the performance will be impacted negatively because of the overhead of each data loading.

C. Scalability

Strong scaling tests were conducted to show the scalability of our algorithm. We measured the total time for running our algorithm, including time for stream surface integration, communication and runtime seeding curve segment subdivision. Disk I/O time was not included, because it was done in the initialization stage through the library BIL [15]. For all four of our datasets, the cache size was 16. The static cutting threshold and the runtime cutting threshold were 16 and 128 for *Isabel*, *MJO*, and *Nek* respectively. For *Plume*, they were set at 16 and 32. For *Isabel*, up to 1K processes were used. The maximum number of integration steps was 400, and 25^3 block size was used. For *MJO*, up to 4K processes were used. The maximum number of integration steps was 200 and the block size was 25^3. For *Plume*, up to 8K processes were used. 900 was the maximum number of integration steps and the block size was 32^3. For *Nek*, up to 8K processes were used. The maximum number of integration steps was 650, and the block size was 32^3.

The results of strong scaling tests are shown in Figure 12. The top row shows the scalability of our algorithm. The results demonstrate a good scalability of our algorithm up to a large process count. The bottom row shows the percentage of time spent on different components: computation, communication and runtime seeding curve segment subdivision. Overall, a majority of the time was spent on computation. For all of the four datasets, the time for seeding curve segment subdivision is negligible. Also, as the process count increases, the percentage of communication increases and this is because with a large number processes, the time spent on work stealing increases.

D. Load Balancing

As described in Section III-C3, work stealing and runtime seeding curve subdivision are employed in our algorithm to balance the computational workload of different processes. To demonstrate the effectiveness of our load balancing algorithm, we computed the relative percentage imbalance under different process count and the results are shown in the top row of Figure 13. Perfect computational load balancing was observed for *Isabel*, *MJO* and *Nek*. For *Isabel* and *MJO*, the relative percentage imbalance was below 5% for up to 1K and 4K processes. For *Nek*, the imbalance was below 10% for up to 8K processes. *Plume* has a relative percentage imbalance value below 10% for up to 4K process. When 8K processes were used for *Plume*, the relative percentage imbalance increased to around 40% and this was because there were not enough tasks for this large number of processes. We also measured the computation time for each process when 1K processes were used for *Isabel* and 4K processes for *MJO*, *Plume*, and *Nek*. The results are shown in the bottom row of Figure 13. The small difference in the computation time among the different processes indicates the effectiveness of our load balancing algorithm.
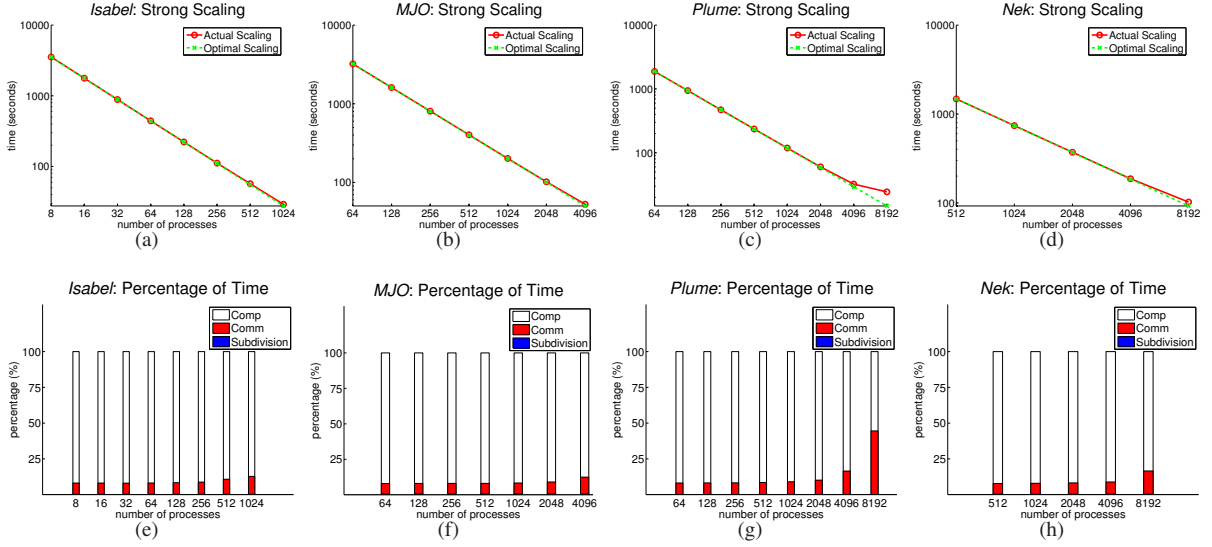
Fig. 12: Strong scaling results. The top row graphs the time for running our algorithm, including time for stream surface computation, communication and runtime seeding curve segment subdivision for different number of processes. The bottom row contains the percentage of time spent on each component.
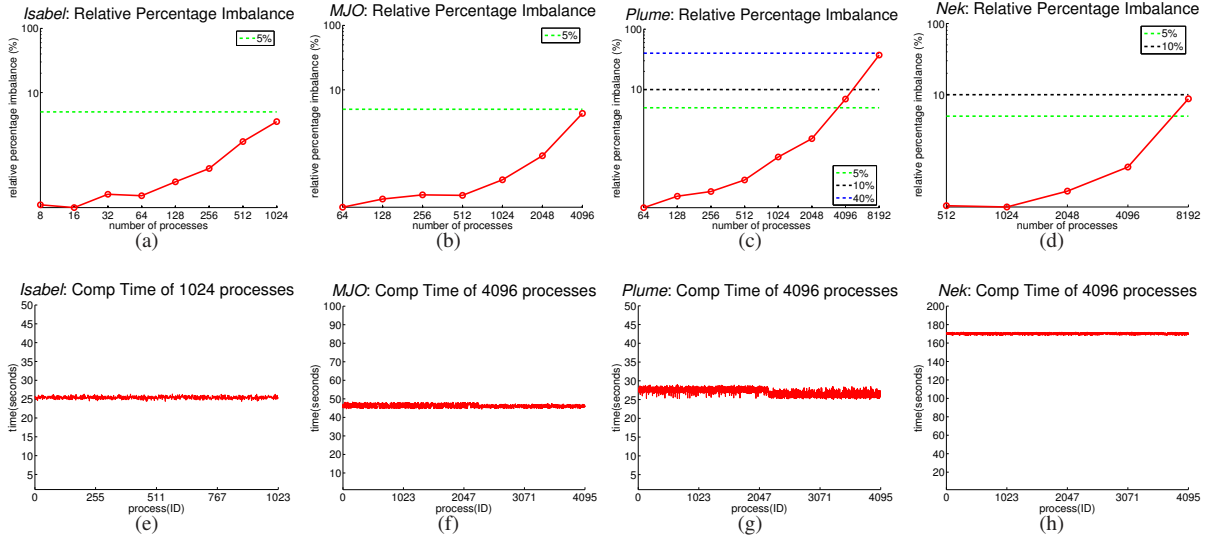

Fig. 13: The top row shows the relative percentage imbalance under different process count. The bottom row contains the computation time for each process when 1K processes were used for *Isabel* and 4K processes were used for *MJO*, *Plume*, *Nek*.

V. CONCLUSION AND FUTURE WORK

In this paper, we present a new algorithm for parallel stream surface computation. By partitioning seeding curves into segments, stream surfaces are split into small stream surface patches, and processes integrate different stream surface patches in parallel. Several strategies are applied to improve the overall performance of our algorithm. Loading data from other processes instead of from disk reduces the I/O cost. Cache is used to reduce the number of data loads. Runtime seeding curve segment subdivision and work stealing is used to improve the load balancing of our algorithm dynamically. Several experiments were conducted to study different pa-

rameters' influence on the performance of our algorithm and their relationships. Based on these experiments, we provided some guidance for parameter setting. We also demonstrated the scalability of our algorithm up to a large number of processes.

In the future, we plan to extend our algorithm to unsteady flow fields. For time-varying flows, the memory requirement would increase because multiple time steps are involved. The constraint in the amount of memory will prevent us from loading all data blocks to memory in the initialization stage, and hence a more complicated I/O strategy will be required.

REFERENCES

[1] D. Camp, C. Garth, H. Childs, D. Pugmire, and K. Joy, "Streamline integration using mpi-hybrid parallelism on a large multicore architecture," *Visualization and Computer Graphics, IEEE Transactions on*, vol. 17, no. 11, pp. 1702–1713, 2011.

[2] D. Pugmire, H. Childs, C. Garth, S. Ahern, and G. Weber, "Scalable computation of streamlines on very large datasets," in *High Performance Computing Networking, Storage and Analysis, Proceedings of the Conference on*, 2009, pp. 1–12.

[3] B. Nouanesengsy, T.-Y. Lee, and H.-W. Shen, "Load-balanced parallel streamline generation on large scale vector fields," *Visualization and Computer Graphics, IEEE Transactions on*, vol. 17, no. 12, pp. 1785–1794, 2011.

[4] T. Peterka, R. Ross, B. Nouanesengsy, 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 the 2011 IEEE International Parallel & Distributed Processing Symposium*, ser. IPDPS '11, 2011, pp. 580–591.

[5] D. Camp, H. Childs, C. Garth, D. Pugmire, and K. Joy, "Parallel stream surface computation for large data sets," in *Large Data Analysis and Visualization (LDAV), 2012 IEEE Symposium on*, 2012, pp. 39–47.

[6] J. P. M. Hultquist, "Constructing stream surfaces in steady 3d vector fields," in *Visualization, 1992. Visualization '92, Proceedings., IEEE Conference on*, 1992, pp. 171–178.

[7] C. Garth, X. Tricoche, T. Salzbrunn, T. Bobach, and G. Scheuermann, "Surface techniques for vortex visualization," in *Proceedings of the Sixth Joint Eurographics - IEEE TCVG Conference on Visualization*, ser. VISSYM'04, 2004, pp. 155–164.

[8] T. McLoughlin, R. S. Laramee, and E. Zhang, "Easy integral surfaces: A fast, quad-based stream and path surface algorithm," in *Proceedings of the 2009 Computer Graphics International Conference*, ser. CGI '09, 2009, pp. 73–82.

[9] D.Sujudi and R.Haimes, "Integration of particles and streamlines in a spatially-decomposed computation," in *Proceedings of Parallel Computational Fluid Dynamics*, 1996.

[10] W. Kendall, J. Wang, M. Allen, T. Peterka, J. Huang, and D. Erickson, "Simplified parallel domain traversal," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '11. ACM, 2011, pp. 10:1–10:11.

[11] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[12] J. van Wijk, "Implicit stream surfaces," in *Visualization, 1993. Visualization '93, Proceedings., IEEE Conference on*, 1993, pp. 245–252.

[13] T. Stter, T. Weinkauf, H.-P. Seidel, and H. Theisel, "Implicit integral surfaces," in *Proc. Vision, Modeling and Visualization*, November 2012, pp. 127–134.

[14] T. Peterka, R. Ross, A. Gyulassy, V. Pascucci, W. Kendall, H.-W. Shen, T.-Y. Lee, and A. Chaudhuri, "Scalable parallel building blocks for custom data analysis," in *Large Data Analysis and Visualization (LDAV), 2011 IEEE Symposium on*, 2011, pp. 105–112.

[15] W. Kendall, M. Glatter, J. Huang, T. Peterka, R. Latham, and R. Ross, "Terascale data organization for discovering multivariate climatic trends," in *High Performance Computing Networking, Storage and Analysis, Proceedings of the Conference on*, 2009, pp. 1–12.

[16] J. Dinan, D. B. Larkins, P. Sadayappan, S. Krishnamoorthy, and J. Nieplocha, "Scalable work stealing," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, ser. SC '09, 2009, pp. 53:1–53:11.

[17] R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing," *J. ACM*, vol. 46, no. 5, pp. 720–748, 1999.

[18] N. Francez, "Distributed termination," *ACM Trans. Program. Lang. Syst.*, vol. 2, no. 1, pp. 42–55, 1980.

[19] E. Merzari, W. Pointer, A. Obabko, and P. Fischer, "On the numerical simulation of thermal striping in the upper plenum of a fast reactor," *Proceedings of ICAPP*, 2010.