Graph-based Seed Scheduling for Out-of-core FTLE and Pathline Computation

Chun-Ming Chen Han-Wei Shen

The Ohio State University *

ABSTRACT

As the size of scientific data sets continues to increase, performing effective data analysis and visualization becomes increasingly difficult. Desktop machines, still the scientists' favorite platform to perform analysis and visualization computation, usually do not have enough memory to load the entire data set all at once. For time-varying flow visualization, the Finite-Time Lyapunov Exponent (FTLE) allows one to glean insight into the existence of the Lagrangian Coherence Structures (LCS) by quantifying the separation of flows. To obtain high resolution FTLE fields, the computation of FTLE requires tracing particles from every grid point and at every time step. Because the size of the time-varying flow data can easily exceed the amount of available memory in the desktop machines, efficient out-of-core FTLE computation algorithms that minimize the I/O overhead are very much needed. To tackle this problem, one can perform a batch mode computation of particle tracing where the particles are organized into different groups, and at any time only one group of particles are advected in the time-varying field. Since tracing particles requires loading the necessary data blocks on demand along the flow paths, to maximize the usage of the data and minimize the I/O cost, an effective scheduling of particles becomes essential. The main challenge is to avoid reloading the same data blocks that were previously processed. In this paper, to solve the problem we model the flow as a directed weighted graph and predict the access dependency among the data blocks, i.e., the path of particles, using Markov chain. With the predicted path we devise an optimization method that groups the particles into different processing batches to minimize the total number of block accesses from the disk. Experimental results show that our scheduling algorithm performs better than algorithms based on a general space-filling ordering.

1 INTRODUCTION

The rapid growth of computation power has enabled scientists to generate large data sets at a very high resolution. Although large scale simulations are usually done on supercomputers, scientists still prefer to carry out the data analysis and visualization tasks on their own desktop machines whenever possible. Thanks to the improved processor speed and more sophisticated memory and storage systems, performing visualization of large data sets on desktop computers is now much more plausible than before.

Flow visualization plays an important role in many scientific and medical disciplines. Among several existing flow visualization techniques, pathlines are commonly used for the visualization of time-varying flow data. Pathlines are the trajectories of particles in the flow field over time, the computation of which can be used as a fundamental technique to produce other visualization such as streak lines, path surfaces as well as the Finite-Time Lyapunov Exponent (FTLE) fields.

*e-mail: {chenchu,hwshen}@cse.ohio-state.edu

IEEE Symposium on Large Data Analysis and Visualization 2013
October 13 - 14, Atlanta, Georgia, USA
978-1-4799-1658-0/13/$31.00 ©2013 IEEE

In this work, our goal is to compute high resolution FTLE fields from densely seeded pathlines on a multicore desktop machine. The FTLE field is a scalar field recently proposed by Haller [12] to quantify the separation rate of neighboring flow trajectories in a time-varying flow field over a finite time interval, which is very effective for depicting the separation of flows that are not easily seen by directly inspecting the velocity field itself. Although the FTLE field can reveal important structures, its computation cost is much higher than most visualization techniques that only involve local computation. This is because to measure the separation rate of flow in a local region, it is necessary to compute the *flow map*, by tracing particles densely in the domain, e.g., from every grid point. Moreover, particles are seeded at every time step to obtain a sequence of FTLE fields. Many methods have been proposed to speed up the flow map generation [22, 10, 2, 13]. Without trading accuracy for speed, some methods compute densely seeded pathlines in parallel using hardware acceleration such as GPGPU [8, 11] or clusters [17]. While these methods provide out-of-core strategies to achieve high throughput by overlapping I/O and computation, most of them assume that the intermediate particle locations can be stored in memory during the flow map computation. As the datasets become larger, however, this assumption is not valid any more, because the particles seeded in every grid point and every time step of the data will also require large amount of memory storage.

To tackle the issue of processing excessive amount of particles, one can perform batch mode computation by advecting a smaller group of seeds at a time, under the memory space constraint. The whole flow map computation is thus divided into a number of rounds, where in each round only a selected group of seeds are stored in memory and advected until they meet the termination condition. To achieve efficient I/O, pathlines are computed out-of-core, i.e., partial data are loaded in memory on demand during the computation. Generally, efficient out-of-core computation requires full utilization of data while they are in memory. Since particles from different seeding locations and time steps may converge or follow similar paths, in order to prevent the same data from being loaded multiple times, a good scheduling scheme that determines the grouping of particles to be advected together is needed.

To maximize the data usage and minimize the I/O cost for out-of-core FTLE computation, in this paper we use a graph-based approach which can effectively model the local flow behavior in neighboring regions in both space and time. With a graph created from the flow directions, we analyze the data dependency during particle tracing for the entire time-varying flow field using discrete-time Markov chains. After the global data dependency is predicted, we group similar paths together and schedule the seeds accordingly to perform batch computation of FTLE. Experiments show that our scheduling can achieve fewer disk accesses compared to a general space-filling curve ordering of particle advection.

The structure of the paper is as follows. The next section introduces FTLE and reviews the related works. Section 3 discusses the challenge of out-of-core FTLE computation and we propose a scheduling algorithm in Section 4. The out-of-core system is presented in Section 5, followed by the experimental results and discussions in Section 6. We conclude the paper in Section 7.

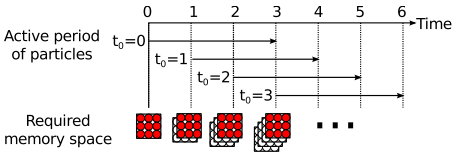

Figure 1: Illustration of classic out-of-core particle tracing strategy for flow map generation. Assuming the integration step 4, starting from the 4^{th} time frame the system has to store all particle locations from the current (red circles) and previous 3 time steps (white circles) in memory.

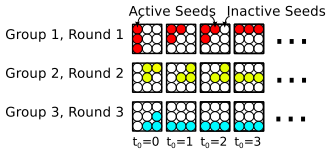

Figure 2: With limited memory capacity for storing active particles, the initial seeds are divided into several groups and advected separately per group. A group can contain seeds from different time steps.

2 Background and Related Works

In this section we provide a brief overview of FTLE and discuss the related techniques to speed up its computation.

Finite-time Lyapunov exponent (FTLE). The Finite-time Lyapunov Exponent is a scalar value which measures the separation of flow in its neighborhood along a finite time period. Haller [12] shows that the ridges of the FTLE field can be used to identify the Lagrangian coherent structures (LCS). The LCS divides distinct regions in the dynamical system and is useful for the visualization and analysis of transport barriers [24]. Pobitzer *et al.* provide an overview of the techniques for topology-based visualization of unsteady flow fields [19].

The calculation of FTLE depends on an accurate measurement of flow divergence, which is computed by tracing particles over an infinitesimal distance for each point of interest. Given the integration interval, we generate a mapping between the start and end positions of particle transportation, usually called the *flow map*. Formally, given a time-varying velocity field $\mathbf{v} = f(\mathrm{x},t)$ with $f : \mathbb{R}^n \times \mathbb{R} \to \mathbb{R}^n$, its flow map is defined as

$$\phi_{t_0}^{t_0+T} : \mathbb{R}^n \to \mathbb{R}^n,\ \mathbf{x}_0 \mapsto \phi_{t_0}^{t_0+T}(\mathbf{x}_0) = \mathbf{x}(t_0+T;t_0,\mathbf{x}_0), \quad (1)$$

where $\mathbf{x}(t_0+T;t_0,\mathbf{x}_0)$ is the location of a particle trajectory (or pathline) starting from point $\mathbf{x}_0$ and time t_0 after traveling a time interval T under the velocity field. Since we consider the differences of trajectories within an infinitesimal neighborhood, the gradient of the flow map $\nabla\phi_{t_0}^{t_0+T}$ is used to form the *right Cauchy-Green deformation tensor*, defined as

$$\Delta_{t_0}^{t_0+T} = \left(\nabla\phi_{t_0}^{t_0+T}\right)^* \nabla\phi_{t_0}^{t_0+T} \quad (2)$$

where $(\cdot)^*$ represents matrix transpose. The square root of its maximal eigenvalue, $\sqrt{\lambda_{max}(\Delta_{t_0}^{t_0+T})}$ corresponds to the maximal local change in distance due to deformation. Taking the exponent, FTLE is defined as

$$\sigma_{t_0}^{t_0+T} = \frac{1}{T}\ln\sqrt{\lambda_{max}(\Delta_{t_0}^{t_0+T})}. \quad (3)$$

The FTLE values depend on the input starting time t_0 and integration time T. Usually an appropriate integration time T is assigned by the scientists and the FTLE fields regarding every starting time t_0 is computed.

In practice, in order to generate high resolution FTLE fields, it is necessary to construct the flow map by seeding and tracing particles as densely as possible. Since this computation requires extremely large amounts of processing time, several accelerative approaches have been proposed. One strategy is to approximate the flow map by adaptively reducing the spatial seeding density based on the surrounding flow divergence. Garth *et al.* [10] present an adaptive refinement algorithm to approximate FTLE given an error constraint. Sadlo *et al.* filter the computation of FTLE on expected ridge regions [22], and a grid advection technique to track the ridges is then proposed [23]. Kasten *et al.* [15] introduce *localized* FTLE that reuses the FTLE values from the previous steps. To gain further speed up, though with more error introduced, Brunton and Rowley [2] and Hlawatsch *et al.* [13] both present similar ideas to approximate the flow map by hierarchically interpolating short integrals over time.

As parallel computation platforms become more pervasive, instead of reducing the sampling seeds, another strategy is to use GPUs and APUs (Accelerated Processing Units) to compute FTLE in parallel for mid-scale datasets [8, 11], or to use HPC platforms to handle very large datasets [17].

Out-of-core particle advection. Designing efficient out-of-core algorithms has been an active area in research of large-data visualization. Silva *et al.* [25] provide a comprehensive review of out-of-core visualization techniques. Cox and Ellsworth [9] propose to improve data locality by storing data in spatially coherent blocks and to use application-controlled demand paging over system paging for better I/O performance. Sulatycke and Ghose [26] propose a multi-threaded out-of-core system to reduce IO overhead. For out-of-core particle advection computation, Pugmire *et al.* [20] use a hybrid of static block allocation and load-on-demand strategy to achieve load balance and minimum I/O on supercomputers. Camp *et al.* [3] cached flow data in solid state drives to reduce the overhead of repeated I/O. To visualize particle traces in real-time, Bruckschen *et al.* [1] compute and store pathlines in the preprocessing stage and load the precomputed particle traces according to the input.

Graph modeling for succinct representation of flow fields has become popular recently. Chen *et al.* [6] use the Morse Connection Graphs model to represent the topological structure of the flow fields. Xu and Shen [27] propose a node-link graph model for flow field exploration. Nouanesengsy et al. [18] model the flow field as a graph to predict and balance the runtime workload for parallel streamline computation. Chen *et al.* extend the flow graph with multiple hops to compute I/O-friendly file layouts for static [5] and time-varying flow fields [4]. Reich and Scheuermann [21] approximate FTLE at infinity by applying Markov chains on static unstructured flow fields encoded into a graph representation. Ma *et al.* [16] recently propose a hierarchical graph-based modeling and visualization method for flow field exploration and query. Chen and Fujishiro [7] optimize the domain partitioning of unstructured flow fields for parallel streamline computation. The flow field is modeled as a graph based on an anisotropic local diffusion operator and partitioned with a multilevel spectral graph bisection method that minimizes the sum of edge weights among the partitions. While their algorithm shares similar goal of domain partitioning as ours, we employ Markov chains to encode long-term flow behavior and use this prediction for later scheduling and evaluation.

3 Proposed Framework

In this section we discuss the practical challenge for computing high-resolution FTLE fields with limited memory capacity. Then we briefly give an overview of our proposed particle advection framework to tackle the problem.

3.1 Challenge for parallel FTLE computation on a memory-limited system

To speed up FTLE computation, the existing hardware accelerated methods [8, 17] in general divide the pathline computation into a number of processing units and maximize processor utilization by overlapping the computation and I/O. Since the data can be big, without waiting for the entire flow field to be loaded from the secondary storage device to the memory, most methods use the strategy that loads and advects particles iteratively from the first time step to the last during forward particle tracing. Since particles will never go backward in time, once all particles have passed a time step, the corresponding flow field data can be released for future time steps. Double-buffering can be used during the computation - one for particle advection in the current time step, and the other for the I/O thread to preload data in the next time step. Figure 1 illustrates this strategy. This method is I/O efficient because each time step of data will be loaded only once for the entire computation.

Despite its I/O efficiency, the previous methods are in fact difficult to scale. As shown in Figure 1, this type of methods assumes all the intermediate particle positions are stored in memory. Storing the active particle positions, however, will require many times the size of a time step of the flow field. For example, to generate a flow map with 20 integration steps for all time steps, the number of active particles from time steps 1 to time step 20 will continue to accumulate till the 20^{th} time step. For a dataset of dimension 500^3, around 40 gigabytes of memory will be needed to store all the temporary particles in the format of vectors in single-precision floating points, which far exceeds the memory capacity of a normal desktop computer.

Since particles in the entire domain cannot be traced at the same time due to the limited memory capacity in a desktop computer, we can only allow a smaller portion of the seeds to be resident in memory at a time. For this reason, it is important to break the particles into groups and schedule the execution of the particles carefully to maximize the computation performance and minimize the I/O overhead. Below we present the main idea of our algorithm.

3.2 Proposed Framework

Because of the memory space limitation, we can only trace a small group of seeds at a time. We call the advection of such a seed group a *round*. Since in each round not all the flow field data are to be used, the particle advection is performed in an out-of-core manner. We decompose and store the data in spatially coherent blocks and manually manage the block I/O in an application-controlled memory pool, as suggested by Cox and Ellsworth [9]. For time-varying datasets, we first uniformly divide the spatial domain into blocks and further increase the data coherence by merging several blocks in consecutive time steps from the same spatial location together. Since these merged space-time blocks are the unit data block for each disk access in our system, we will simply refer them as a *block* hereafter in the paper.

Although our pathline computation framework has a memory pool to cache the loaded data blocks, we cannot assume those cached blocks remaining in one round will be reused in other rounds. Since reloading the same data block multiple times during the computation increases the number of disk I/O accesses and thus reduces the I/O efficiency, a good grouping and scheduling of seeds for each round is an important factor for the overall performance of out-of-core FTLE computation. Therefore in this work we employ a preprocessing stage to schedule the seeds for all rounds, as described below.

The seed scheduling is determined before flow map computation. It is a quick preprocessing aiming at reducing the potential number of disk accesses at run time. To do this, as how flow data are segmented into blocks, seeds in the seeding domain are also first divided into blocks, defined as *seed blocks*. In each round the flow map computation processes a group of seed blocks. The benefit for using the seed blocks is that it reduces the problem size for scheduling analysis, and the flow map generated in blocks is easier to store and index. For simpler analysis and implementation, we partition the seed blocks in the same partitioning as the flow data blocks. Since the blocks are small, we can assume that all seeds in the same seed block have similar trajectories and thus they will visit similar data blocks.

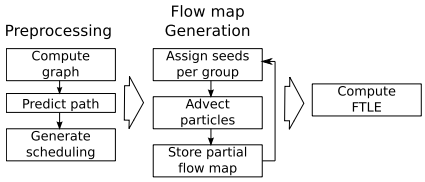

Figure 3: An overview of our out-of-core FTLE computation framework.

Now the next goal is to determine the optimal grouping of the seed blocks, where all seeds in one group will be advected in the same round. The basic idea is that the seed blocks that share similar flow trajectories will be grouped together, so as to reduce the number of data blocks to load in each round. The method to predict the particle trajectories from each seed block will be discussed in section 4.1. The seed group size is controlled by available memory size designated for storing the active seeds during particle advection. Figure 2 illustrates an example of grouping the initial seeds over different time steps.

In addition to determining the optimal grouping of the seed blocks, we also need to determine the computation order of the groups to maximize the reuse rate of data blocks by consecutive rounds once they are loaded in the memory. To achieve this, we design an algorithm that combines the goals of optimal grouping and scheduling together to ensure a high degree of data reuse without assuming the group size. The detail will be described in section 4.2. The overview of our out-of-core FTLE computation framework is shown in Figure 3.

4 Scheduling Algorithm

The goal for scheduling seed advection is to reduce the number of data accesses and thus increase the performance of flow map generation. The idea is to minimize the number of blocks being reloaded due to memory misses. In the following section 4.1, we describe how we predict the set of data blocks along the advection path of each seeding block, utilizing a graph model and Markov chains. With the path information and seed assignment for each round, the total number of data blocks to load can be estimated. We minimize this total data block accesses with an approximated optimal seed grouping scheme that works well for all possible grouping sizes, as described in section 4.2.

4.1 Flow modeling and path prediction

To minimize the total number of data block accesses from different rounds of particle advection, it is necessary to have the information about what blocks a particle will visit from each seeding location. The naive way to obtain the exact path is through tracing all seeds with full integration steps, but the costly computation does not serve our purpose in the preprocessing stage. In order to save time, we compute the flow trajectories in local regions and use the information to predict the global flow behavior.

Flow modeling. As stated in Section 3.2, during the out-of-core flow map generation the time-varying flow field is decomposed into blocks over all time steps. Since our goal is to minimize the number of block accesses, the flow field can be analyzed and modeled

at the granularity of blocks. We model the inter-block dependencies of the flow field as a directed weighted graph, by adopting the access dependency graph for modeling time-varying flow fields by Chen *et al*. [4]. The graph modeling regards the time-varying flow field as a 4D dataset and aggregates all blocks from all time steps into the graph. In the graph, each node represents a block. The number of nodes in the graph equals to the number of blocks in the time-varying flow field. The local access dependency between each pair of spatially or temporally adjacent blocks is modeled as a graph edge connecting the corresponding nodes. The edge weight is assigned as the probability of uniformly distributed particles moving from one block to the other (in different space or time). To compute the edge weight, particles are uniformly sampled in each block and the percentage of the particles moving to the neighboring blocks is collected. The computation for graph generation only involves particle advection within each block, and thus can be easily parallelized and efficiently computed.

Prediction by discrete-time Markov chains. We use the flow graph G to model the access dependencies between neighboring blocks as described above to predict the block-wise access dependencies along the particle paths for the entire flow field. Since in G the sum of all outgoing edge weights for each node is either one or less than one (when particles move into a sink or to the domain boundary), the graph weight $w(i, j)$ can be seen as the transition probability of a random walk (or a Markov chain) from node i to node j. That is, if there are particles uniformly distributed in block i, then in the next *step*, particles at the percentage $w(i, j)$ will move to block j. We can propagate the particles further using discrete-time Markov chains, by assuming that the probability of particles moving from one node to another in any step only depends on its previous step. By repeatedly propagating the probabilities through Markov chains from each initiating block, we can collect the visited nodes as the predicted dependent blocks related to their initiating block. The collected predicted paths are then used in our scheduling algorithm described later. Below we first describe the prediction algorithm.

The flow graph G with n nodes is first represented as an $n \times n$ adjacent matrix A. We use the format of column-stochastic matrix, where the j^{th} column of A represents the transition probabilities *out* of the vertex v_j, and thus A_{ij} represents the weight of edge from node j to node i in the graph G. We also denote the percentage of particles distributing among all nodes as an $n \times 1$ column vector x, where x_i represents the percentage of particles in node i. Note that the sum of values in vector x in any step is always less or equal to one:

$$\sum_{1 \le i \le n} x_i \le 1. \tag{4}$$

The propagation of particles among the nodes from step s to step $s+1$ can be written as a matrix-vector multiplication:

$$x^{s+1} = Ax^s. \tag{5}$$

We can also stack several different column vectors together to form a probability distribution matrix X and propagate the particle distribution at once using a matrix-matrix multiplication:

$$X^{s+1} = AX^s. \tag{6}$$

Essentially, the j^{th} column vector X_j^{s+1} represents the particle distribution propagated from X_j^s.

Since the final goal is to collect the dependent blocks with particles initiating from each seeding block separately, we assign the initial stacked probability distribution matrix as an $n \times n$ identity matrix, i.e.,

$$X^1 = I_{n \times n}. \tag{7}$$

In other words, we assign full probability (one) in distinct nodes for each column vector of X, and then iteratively propagate the distribution using Equation 6. After several steps of propagation, the j^{th} column vector X_j^s will represent the probability distribution of particles initiating from block j, since initially $X_{jj}^1 = 1$.

To collect the block dependencies, during each iteration we locate the non-zero elements X_{ij} in the matrix X, indicating that node i will be visited from node j in some step. To record these visited nodes we create an empty $n \times n$ matrix V and assign $V_{ij} = 1$ when X_{ij} is not zero. After running several steps the matrix V represents the access dependency among nodes within these steps where each element V_{ij} in the j^{th} column of V represents whether the corresponding node i will be visited from node j.

To prevent the predicted access dependency matrix from collecting blocks which have a tiny probability to be visited from the initial block, we use a threshold th and only collect the blocks which has the visiting probability larger than th. In other words, we assign $V_{ij} = 1$ only when $X_{ij} > th$. The thresholding reduces the resulting size of matrix V and only keeps the dependent blocks of higher confidence. The threshold value should be small in order to collect sufficient probable dependent blocks, but too small of a threshold value will require large storage space for the matrix V. The selection of the threshold parameter will be discussed in the experiment section. Algorithm 1 shows the pseudocode of the final prediction algorithm.

Algorithm 1 Discrete Markov-chain Path Prediction Algorithm

Require: Adjacent matrix A in size $n \times n$, threshold th.
1: Let X be an identity matrix in size $n \times n$.
2: Let V be an empty matrix in size $n \times n$.
3: **while** V not converge or exceed maximum iterations **do**
4: Propagate: Let $X = AX$
5: For each element X_{ij} in X, set $V_{ij} = 1$ if $X_{ij} > th$.
6: **end while**
7: **return** Matrix V as the prediction result.

4.2 Seed Scheduling

After the prediction of block dependencies is done, we use a seed-scheduling algorithm to determine the grouping of seeds for each round and their computation order. The goal is to minimize the number of block accesses at run time. Below we first provide the cost function, which estimates the number of block accesses, and then present our optimization algorithm.

4.2.1 Grouping cost function

With the result of path prediction from the algorithm presented in the previous section, the cost function takes the grouping of seed blocks as the input and returns the total number of blocks to be accessed from the disk. We assume that the memory pool to store the data is large enough for the seeds in the same group to advect together for one time step without memory misses, i.e., no block will be released due to memory full and then reloaded due to revisit of particles in that time step. Since particles will never go back to the previous time steps, an easy choice of the data pool size is as large as the dataset size in one time step. A better choice of the data pool size will be discussed later in this section.

We also assume that the data pool is not big enough for any previously loaded data blocks to be reused for the later rounds. This is reasonable since the given integration step is usually long and thus a large number of blocks will be visited per round. Therefore the number of blocks to load in each group is the number of *distinct* blocks to be visited by the seed group, which can be obtained from the block visiting matrix V described in the previous section. Finally the total number of blocks to load for the entire process is the

sum of the number of the counted distict blocks from each group. We formulate the cost function as follows:

We use P to describe the input grouping as a list of k sets, where k is the number of groups. Let P_i contains a set of indices to the seeding blocks in group i. We denote each element in P_i as $P_{i1}...P_{im}$ where m is the number of elements in P_i. That is,

$$P_i = \{P_{i1}, P_{i2}, \ldots P_{im}\}, 1 \leq i \leq k. \quad (8)$$

Note that each index only appears once across the entire set of groups. Therefore the set elements in all groups are mutually exclusive, i.e.,

$$P_i \bigcap P_j = \emptyset \qquad \forall i, j < k \wedge i \neq j. \quad (9)$$

In Section 4.1 we have formulated an $N \times N$ matrix V which predicts the paths from every node. Recall that N corresponds to the number of seed blocks in the dataset. We denote V_j as the j^{th} column vector of V, where a non-zero element in the vector indicates that the corresponding block will be visited by the seeds from block j. Therefore the number of distinct blocks to be visited from a seed group P_i can be counted with the number of ones after merging the corresponding column vectors of V into one vector using the element-wise *logic-or* ($\vee$) operation, i.e.,

$$c(P_i) = sum(V_{P_{i1}} \vee V_{P_{i2}} \ldots \vee V_{P_{im}}). \quad (10)$$

Here *sum* counts the number of ones in the resulting vector. Finally given the grouping P, the cost representing the total number of blocks to load is

$$cost(P) = \sum_{P_i \in P} c(P_i). \quad (11)$$

Note that the visiting matrix V generated from the previous section records the dependent blocks where particles will visit from each seeding block at maximum steps. In our implementation, the actual integration step is constrained by the user-given parameter T. To compute a precise cost for the given T, after the matrix V is generated, we can simply zero out the matrix elements whose corresponding data blocks are farther than their initiating blocks by T time steps. Formally, given the matrix V and integration steps T, we assign $V_{ij} = 0$ when $Timestep(i) > Timestep(j) + T$, where $Timestep(.)$ returns the corresponding time step for the input block index.

As stated before, the group size, or the number of seed blocks per group, is limited by the memory space allocated for particle advection. Once the seed pool size is given, the group size will be equal to the seed pool size in bytes divided by the memory space to store the profiles of a block of seeds, including their positions. However in practice it is not trivial to determine the seed pool size because with limited memory space the size of the seed pool is competing with that of the data pool. We prefer the seed pool as large as possible so that fewer rounds are needed to process the entire seeds. Meanwhile to fulfill the assumptions about the size of the data pool presented earlier, the data pool size should be large enough to prevent memory misses within each round. Fortunately, the minimum data pool size required for each round can be estimated through the predicted data block dependencies, supposed that the grouping of the seed blocks is determined. However, to optimize the grouping we need to provide the group size. As stated earlier, the group size is determined by the memory space available, which is affected by the unknown data pool size.

To solve this problem, instead of optimizing the grouping for a fixed group size, we generate a hierarchy of grouping where in each level the cost of the respective grouping is optimized. In this way the user can search for the setting of both data and seed pools in the memory space constraint with the minimum cost of runtime data accesses. In addition, by traversing the hierarchy we can determine the order to process the seed groups. This order serves as our seed scheduling order.

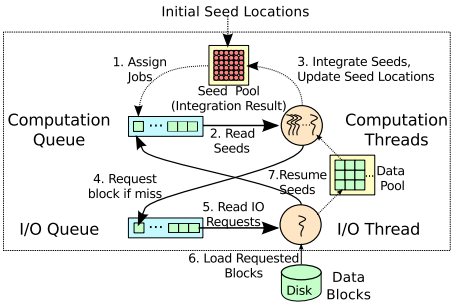

Figure 4: The out-of-core multithreaded particle advection system.

4.2.2 Seed Scheduling algorithm

Our grouping optimization problem is essentially a hierarchical clustering problem that groups seed blocks with similar paths together. As most clustering problems with optimal costs are NP-hard, we use an approximation algorithm to approach the minimum cost. Since our cluster sizes are usually large, building the hierarchy using existing bottom-up hierarchical clustering methods with greedy clustering criteria is not an appropriate solution in our case. Instead, we find the optimal clustering by recursively dividing the seed blocks into two balanced-sized groups, as described below.

We approximate the optimal division by first converting the path prediction matrix back into a weighted undirected graph. Similarly, the graph has N nodes representing the N blocks. Given a pair of nodes, if by the predicted block dependencies there exists a path from one block to the other, we give an edge to the corresponding nodes and assign the weight as the count of intersected blocks between the paths starting from both blocks. Therefore an edge with a lower weight means its two ending nodes will less probably merge in the future, and can be separated. Formally, given the $N \times N$ path prediction matrix V, we create a converted undirected graph G' with N nodes. For each non-zero element V_{ij}, we add an edge (i, j) to G' with weight

$$w(i, j) = sum(V_i \wedge V_j) \quad (12)$$

where $\wedge$ represents element-wise *and* operation and *sum* counts the number of ones in the vector.

Given the converted graph we can now find the minimum balanced cut which results in two disjoint similar-sized subgraphs. The minimum balanced cut essentially separates the seed blocks with minimum common paths between two groups. We use the METIS [14] library, which can efficiently approximate the minimum balanced cut, and recursively find the minimum cut until there are less than two nodes left in the subgraph. During the recursion, a binary tree representing the hierarchy is constructed. Each cut of the graph creates an internal node in the tree and at the bottom level the remaining one or two nodes are added as leaf nodes to the tree.

By traversing the tree in the depth-first search order and outputting the visited leaf nodes, we can obtain the execution order for the seed blocks. By separating the order into k groups, we obtain the seeding blocks to run for each round. Note that we still have the order among groups. Advecting seed groups in this order ensures that the data blocks visited from one group will have high chance to be reused by its sibling group, if they remain in the memory pool. Thus our scheduling method is optimal for different memory size and gives lowest possible number of disk accesses with an as large seed group size as possible.

5 Out-of-core parallel flow map computation

After the order of seed advection is determined by the block grouping result, out-of-core pathline computation is performed, where a

batch of seeds in each group are advected at a time to ensures that there are fewer seeds per round and their temporary positions during the advection can be stored in memory.

We implemented an out-of-core multi-threaded flow map computation system. The system maintains two memory pools, one for the data and one for the intermediate seed positions. The data pool caches the input vector data in blocks, and the seed pool stores the seed locations during the advection computation. There are two types of threads in the system. One I/O thread pages in and out of the data blocks from the disk, and there are several pathline computation threads that advect the seeds and update their locations in the seed pool. The communication among the threads is carried out by a job-queue system. Each type of thread is associated with a queue to manage I/O or computation requests, as illustrated in Figure 4.

At the beginning of the program, the system initiates seeds from the next seed group generated by the scheduling algorithm. The input seeds are divided into separate jobs based on the spatial location of the blocks and the jobs are placed in the computation queue. The computation threads remove jobs from the computation queue and checks the availability of the data blocks needed to advect the seeds. If the data are not available, the seeds will be temporarily suspended and a new I/O request will be placed into the I/O queue. The computation queue keeps advecting seeds when data are available until they move out of the domain boundary or exceed the integration time steps.

The I/O thread loads the corresponding data block into the data pool based on the requests from the I/O queue. If the data pool is full, an unused block will be released. The selection of an unused block will be discussed later in this section. After the data block is loaded, the job with seeds and data ready is then placed back into the computation queue.

The above job-queue system is similar to the one presented by Chen *et al*. [4]. We extend the system to perform multiple rounds with different seed assignments. To further reduce disk I/O cost among the rounds, data in the memory pool from the previous round will remain for the next run in case they can be reused. In addition, in each round to ensure that seeds in the earlier time steps are processed first, for both I/O and computation we use a priority queue that releases jobs from the earliest time step in the queue. When the data pool is full, the I/O thread will start releasing an unused data block from the earlier time steps than the active time step for the seeds, since we know that those blocks will not be used any more. When no data block earlier than the active time steps can be found, one unused block is released in the least recently used manner.

6 Results

In this section, we first describe the datasets used in the experiments and then show the results of both path prediction and the improved performance from our scheduling algorithm. Three datasets were used in our experiment: *Isabel*, *Plume* and *Climate*. *Isabel* is a simulation of the hurricane Isabel from September 2003 over the West Atlantic region. *Plume* is a simulation of the thermal downflow plumes on the surface layer of the sun. Finally, *Climate* represents a climate simulation over the Indian and Pacific Ocean. We took the velocity fields representing the wind or flow speed. The data size of each velocity field is listed in Table 1. The first two datasets is from the National Center for Atmospheric Research and the last is from Pacific Northwest National Laboratory in the United States.

The datasets were first reorganized in time blocks, each of which contains two consecutive timesteps of 32^3 spatial blocks with ghost cells. All the timings of our experiments were measured on a Linux machine with a 7200 RPM disk, 4 GB of RAM and the Intel Core i7-2600 CPU supporting up to eight concurrent threads.

To show how much better our proposed seed-scheduling algorithm performs, we use a four-dimensional Z-curve ordering of seed blocks as a competitive baseline, due to its high data locality.

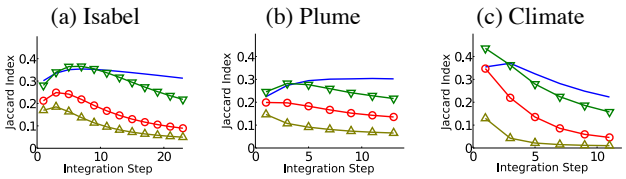

Figure 5: Evaluation of path prediction with threshold values $th = 10^{-4}$ (no mark), 10^{-3} ($\bigtriangledown$), 10^{-2} ($\bigcirc$) and 10^{-1} ($\bigtriangleup$). The Jaccard index shows the similarity between the predicted and the actual path.

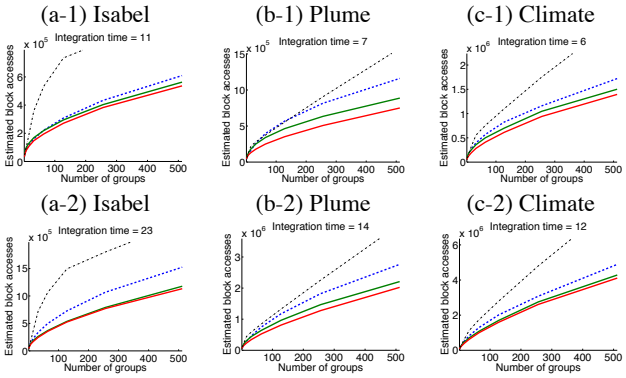

Figure 6: Evaluation of the scheduling by comparing the cost presented in Section 4.2 with different group sizes. Two different integration steps were used: one forth (first row) and a half (second row) of the total time steps of each dataset. In each figure, the curves from the top to the bottom represents for sequential ordering (black dashed), Z-curve (blue dashed) and the scheduling order generated by our algorithm using predicted paths (green solid) and actual paths (red solid).

Given the number of seed blocks to advect per round, determined by the memory space limitation, the Z-curve is uniformly divided into groups where seeds are activated and traced per group. This is the same strategy to schedule seeds from the ordering as described in Section 4.2.2. In addition, in the analysis we also compare with a simple approach by applying a sequential ordering in *xyzt* order with the time domain changing the slowest. This ordering includes the cases that the user simply groups and advects seeds frame by frame.

6.1 Path prediction and evaluation

Our path prediction method requires a one-pass scan of the dataset to generate the access dependency graph. In our experiments, the graph was constructed by performing particle tracing within each time block, where the edge weights were computed by regularly placing one seed per 4^3 voxels at each time step. To generate the dependency graph, we used the parallel out-of-core system described in Section 5, but let all particles terminate at the border of each time block without propagating to the next block. Seven computation threads along with one I/O thread were used. The resulting graph properties and the graph generation time in the experiments are shown on table 1.

The Markov path prediction was implemented and run on Matlab using sparse matrices to save memory usage. To evaluate the quality of the prediction, we generated the actual access dependencies among all blocks by tracing regularly seeded particles from every seed block and gathered the dependent blocks along the particle advection paths. We considered the derived dependent blocks from each seed block as a set. To compare the similarity of the actual and predicted paths from each seed block, we used the Jaccard sim-

Table 1: The dataset properties, the flow graph statistics and each preprocessing time.

Dataset	Dimension	File Size	# Blocks	# Graph Edges	Graph Gen.	Path Gen.	Scheduling Gen.
Isabel	$500^2 \times 100 \times 48$	45.3 GB	48,128	346 K	151 secs	23.2 secs	7.7 secs
Plume	$252^2 \times 1024 \times 29$	54.1 GB	57,344	327 K	676 secs	30.4 secs	19.3 secs
Climate	$2699 \times 599 \times 50 \times 25$	73.1 GB	77,520	974 K	784 secs	136 secs	90.1 secs

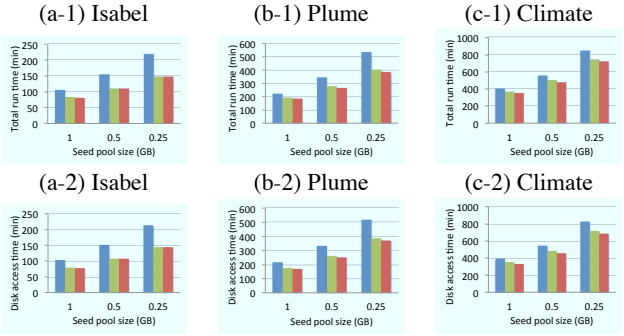

Figure 7: The comparison of total running time (first row) and disk access time (second row) with scheduling using Z-curve (blue bars), our algorithm from predicted paths (green bars), and our algorithm from actual paths (red bars). The integration steps were assigned as half of the total time steps of each dataset.

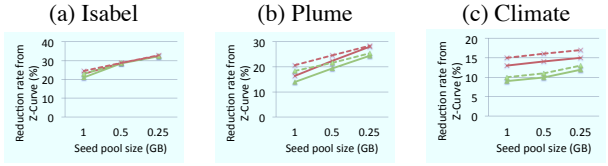

Figure 8: The comparison of total running time reduction (solid curves) and disk access time reduction (dashed curves) using our scheduling algorithm from predicted paths (green curves, marked with $\triangle$) and actual paths (red curves, marked with $\times$) over the Z-curve ordering.

ilarity coefficient, which is the fraction of the intersection over the union of the two sets, i.e,

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|} \tag{13}$$

for two sets A and B. For each integration step, an average Jaccard index of predicted and actual paths from all initiating blocks is computed.

Figure 5 shows the change of the prediction accuracy over increased integration step with different confidence threshold. As can be seen in the figure, by using a lower threshold the predicted paths generally has higher average Jaccard index, meaning that they are more similar to the actual paths. We empirically chose the parameters with $th = 10^{-3}$ in our implementation, since the Jaccard similarity computed using this threshold is close to that of $th = 10^{-4}$ but uses less storage. The generation time of the Markov path prediction using threshold $th = 10^{-3}$, shown in Table 1, was very short compared to the entire flow map generation time.

6.2 Scheduling evaluation

After the dependent blocks for each seed block are computed, the grouping of the seed blocks and the order of pathline computation among them can be determined. To evaluate, we estimate the number of block accesses based on the cost function given in Section

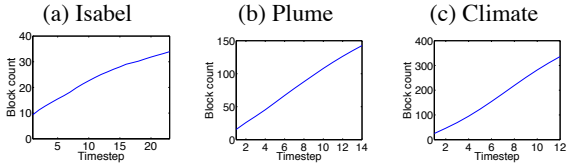

Figure 9: The number of dependent blocks in each time step in the actual path, averaged from each initial seeding block. Higher number of dependent blocks implies that the flow scatters more in the flow field.

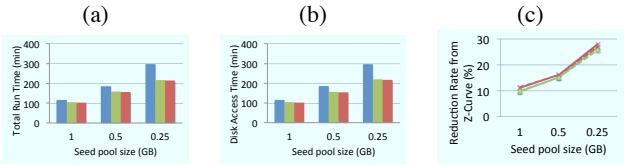

Figure 10: Run-time performance measurement of *Isabel* dataset stored in 64^3 blocks. Similar to Figure 6 and Figure 7, the colors in use represent for Z-curve (blueu), our algorithm using predicted paths (green), and our algorithm using actual paths (red).

4.2, where the path matrix is created using the actual paths collected as stated in the previous section. We compare the scheduling order generated from path prediction with threshold $th = 10^{-3}$ and that from the actual paths. This comparison shows how well the predicted paths can be used to schedule the seed blocks to minimize disk accesses at run time. Four-dimensional Z-curve ordering of the blocks is also compared as a reference for a general but competitive scheduling order, due to its high data locality.

Figure 6 shows the estimated number of block accesses with various numbers of groups and integration steps. The red and green solid curves represent the scheduling results generated by our algorithm using the actual and predicted paths. The blue dashed curve represents the Z-ordering result. As can be seen in the figure, the scheduling generated by our algorithm reduces more disk access times than the Z-curve ordering. Moreover, the result of scheduling by prediction is close to that of using the actual path. With more groups and longer integration steps, the discrepancy of disk accesses between our scheduling and Z-curve becomes larger. This means that the benefit of using our scheduling method is more significant when the dataset is much larger than the available memory and when the integration step is larger.

6.3 Run-time performance

Now we compare the run-time performance of flow map computation using our scheduling scheme with other methods described above. The integration steps in the experiments were assigned as half of the total time steps of each dataset, i.e., 23, 14 and 12 for *Isabel*, *Plume*, and *Climate* dataset, respectively. We tested different memory configurations of the seed pool with the size 1, 0.5 and 0.25 GB, corresponding to 768, 384 and 192 seed blocks per group. Thus the *Isabel* dataset for example was tested with respective 32, 64 and 128 groups of the 24,576 seed blocks from the first 23 time steps. The data pool size was 2GB for all tests.

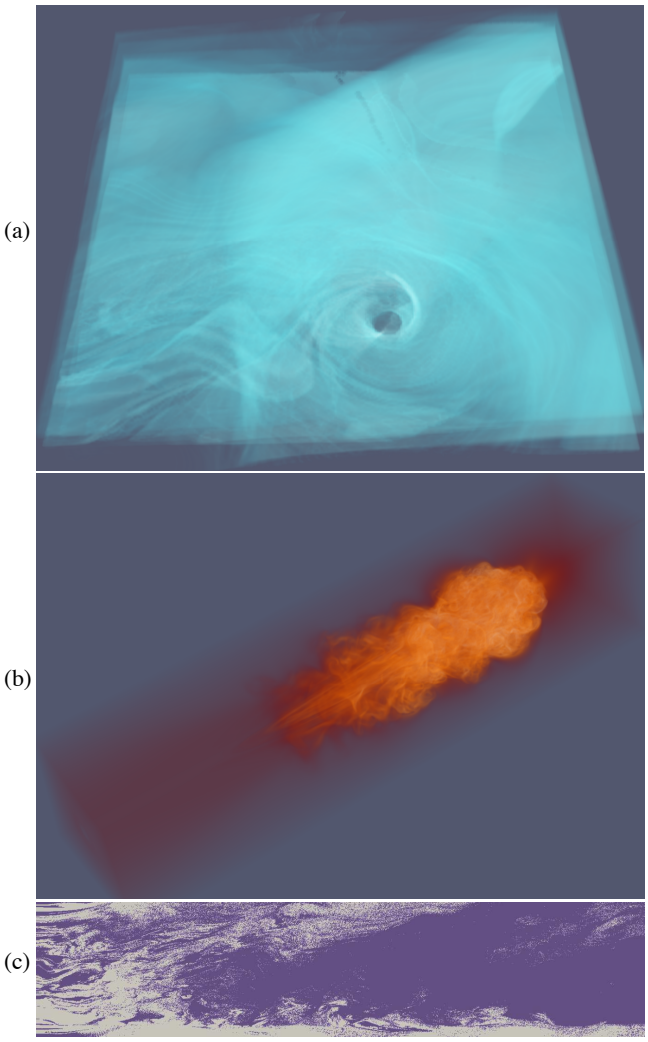

Figure 11: Visualizing the first frame of the FTLE fields for each dataset: (a) *Isabel*; (b) *Plume*; (c): *Climate*, showing one layer on the X-Y plane. The integration step was half of the respective total time steps. Brighter color means higher FTLE value.

The disk access time was computed by summing up the time spent on all disk access function calls done by the I/O thread. As shown in Figure 7, the accumulated I/O time of our seed scheduling from the predicted paths are lower than that of using Z-curve order, and this difference is also shown in the total running time. It can also be seen that when the number of groups becomes larger due to smaller memory pool sizes, the total disk access time becomes larger. Figure 8 shows the time reduction in percentage from the Z-curve ordering. By using our scheduling algorithm (shown as green curves in the figure), we achieved 10% - 33% reduced disk access time and 8% - 32% reduced total running time, over the method using Z-curve ordering. The final FTLE fields are visualized in Figure 11.

6.4 Discussion

As listed in Table 1, the time spent on preprocessing to schedule the computation of pathlines finished within several minutes, far less than the flow map generation time. As shown in Figure 6, our seed scheduling based on the predicted access dependencies can reduce the number of block accesses at a rate very close to that of using the actual pathlines to compute the access dependencies. Comparing the time reduction from using the Z-curve ordering, our scheduling method gains up to 25% speedup, and the time we can save is much larger than the required preprocessing time. This means with a little effort on the preprocessing we can reduce the flow map generation time more. Since the clustering problem is NP-hard, we currently do not have a way to show that our scheduling method is close to the optimal result, and we believe there is still room to improve. We leave the work of clustering optimization as the future work.

Among the three datasets, the *Climate* dataset does not have as much I/O and computation time reduction as the other datasets. We observe a higher divergence of the flow in the *Climate* dataset and hypothesize that the resulting high scattering of the dependent blocks makes clustering the seed blocks more difficult to optimize. Figure 9 shows the number of dependent blocks in each time step in the actual particle traces, averaged from each initial seed block. The *Climate* dataset has the highest block count. In addition, we previously assume that the seeds in the same seed block all move along a similar path. However this assumption may fail when there is a critical point or high divergent flow in the block or along the paths. A finer-grain clustering of the seed blocks may be applied in order to achieve more reduction of the number of disk accesses.

To see the effect of using different block sizes, we tested the performance of *Isabel* dataset stored and scheduled in bigger 64^3 blocks, as shown in Figure 10. In this case our algorithm still outperforms Z-curve ordering but has less time reduction rate compared to the usage of 32^3 blocks in Figure 8(a). This is because our algorithm has less flexibility to schedule the seeds in coarser grains. In addition, the overall running time generally takes longer than that of using 32^3 blocks. Therefore generally it is preferred that the selection of the block size is smaller. However, using a too tiny seed block leads to larger graphs in preprocessing. Note that although in our tests the seed blocks for scheduling have identical sizes as the data blocks, the user may want to use a smaller seed block size when they cannot change the size of data blocks. In this case we can generate the flow graph and apply Markov chains to predict the block dependency with the smaller granularity of the seed blocks, and then merge the graph nodes located in the same data blocks before generating the scheduling order. We leave the improvement of the algorithm as future work.

7 Conclusion and Future Work

In this paper we present an efficient out-of-core seed scheduling algorithm for generating flow maps from large-scale time-varying flow fields, where the available memory is not large enough to hold the entire data set as well as the particle positions. We first model the local access dependencies as a graph and then compute the dependencies for longer integration steps using Markov chains. We provide an algorithm to determine the processing order of the seed blocks using a divide-and-conquer method. We show that our scheduling method outperforms the simpler approach and the scheduling based on Z-curve ordering in both the estimated number of disk accesses and the total flow map generation time. The future work, as stated in Section 6.4, is to further improve the clustering algorithm and reduce the number of disk accesses. For more flexible seed scheduling, the algorithm can be improved to assign different seed block sizes from the given data decomposition size.

Acknowledgements

This work was supported in part by NSF grant IIS-1017635,IIS-1065025, US Department of Energy DOESC0005036, Battelle Contract No. 137365, and Department of Energy SciDAC grant DEFC02-06ER25779, program manager Lucy Nowell. The authors would also like to thank the anonymous reviewers for their comments.

REFERENCES

[1] 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*, pages 45–50, 2001.

[2] S. L. Brunton and C. W. Rowley. Fast computation of finite-time Lyapunov exponent fields for unsteady flows. *Chaos*, 20(1):017503, 2010.

[3] 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*, 17(11):1702–1713, 2011.

[4] C.-M. Chen, B. Nouanesengsy, T.-Y. Lee, and H.-W. Shen. Flow-guided file layout for out-of-core pathline computation. *IEEE Symposium on Large Data Analysis and Visualization (LDAV)*, pages 109–112, 2012.

[5] C.-M. Chen, L. Xu, T. Lee, H. Shen, L. Xu, T.-Y. Lee, and H.-W. Shen. A flow-guided file layout for out-of-core streamline computation. In *Pacific Visualization Symposium (PacificVis), 2012 IEEE*, pages 145–152, 2012.

[6] G. Chen, K. Mischaikow, R. Laramee, and E. Zhang. Efficient morse decompositions of vector fields. *Visualization and Computer Graphics, IEEE Transactions on*, 14(4):848–862, 2008.

[7] L. Chen and I. Fujishiro. Optimizing parallel performance of streamline visualization for large distributed flow datasets. In *Visualization Symposium, 2008. PacificVIS '08. IEEE Pacific*, pages 87–94, 2008.

[8] C. Conti, D. Rossinelli, and P. Koumoutsakos. GPU and APU computations of Finite Time Lyapunov Exponent fields. *Journal of Computational Physics*, 231(5):2229–2244, 2012.

[9] M. Cox and D. Ellsworth. Application-controlled demand paging for out-of-core visualization. In *Vis '97: Proceedings of IEEE Visualization*, pages 235–244, 1997.

[10] C. Garth, F. Gerhardt, X. Tricoche, and H. Hagen. Efficient computation and visualization of coherent structures in fluid flow applications. *Visualization and Computer Graphics, IEEE Transactions on*, 13(6):1464–1471, 2007.

[11] C. Garth, G.-S. Li, X. Tricoche, C. D. Hansen, and H. Hagen. Visualization of coherent structures in transient 2d flows. *TopoInVis '07: Procedings of the Workshop on Topology-Based Methods in Visualization 2007*, pages 1–14, 2007.

[12] G. Haller. Distinguished material surfaces and coherent structures in three-dimensional fluid flows. *Physica D: Nonlinear Phenomena*, 149(4):248–277, 2001.

[13] M. Hlawatsch, F. Sadlo, and D. Weiskopf. Hierarchical line integration. *IEEE transactions on visualization and computer graphics*, 17(8):1148–63, 2011.

[14] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20(1):359–392, Dec. 1998.

[15] J. Kasten, C. Petz, I. Hotz, B. Noack, and H. Hege. Localized finite-time lyapunov exponent for unsteady flow analysis. *Proceedings of the Vision, Modeling, and Visualization Workshop*, 1, 2009.

[16] J. Ma, C. Wang, and C.-k. Shene. FlowGraph : A Compound Hierarchical Graph for Flow Field Exploration. In *Pacific Visualization Symposium (PacificVis), 2013 IEEE*, pages 233–240, 2013.

[17] B. Nouanesengsy, T.-Y. Lee, K. Lu, H.-W. Shen, and T. Peterka. Parallel particle advection and ftle computation for time-varying flow fields. In *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*, pages 1–11, 2012.

[18] 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.

[19] A. Pobitzer, R. Peikert, R. Fuchs, B. Schindler, A. Kuhn, H. Theisel, K. Matković, and H. Hauser. The State of the Art in Topology-Based Visualization of Unsteady Flow. *Computer Graphics Forum*, 30(6):1789–1811, 2011.

[20] 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*, pages 1–12, 2009.

[21] W. Reich and G. Scheuermann. Analysis of streamline separation at infinity using time-discrete markov chains. *Visualization and Computer Graphics, IEEE Transactions on*, 18(12):2140–2148, 2012.

[22] F. Sadlo and R. Peikert. Efficient visualization of lagrangian coherent structures by filtered AMR ridge extraction. *Visualization and Computer Graphics, IEEE Transactions on*, 13(6):1456–63, 2007.

[23] F. Sadlo, A. Rigazzi, and R. Peikert. Time-dependent visualization of lagrangian coherent structures by grid advection. In V. Pascucci, X. Tricoche, H. Hagen, and J. Tierny, editors, *Topological Methods in Data Analysis and Visualization*, Mathematics and Visualization, pages 151–165. Springer Berlin Heidelberg, 2011.

[24] S. C. Shadden, F. Lekien, and J. E. Marsden. Definition and properties of lagrangian coherent structures from finite-time lyapunov exponents in two-dimensional aperiodic flows. *Physica D: Nonlinear Phenomena*, 212(3-4):271–304, 2005.

[25] C. T. Silva, Y.-J. Chiang, J. El-sana, and P. Lindstrom. Out-of-core algorithms for scientific visualization and computer graphics. *Visualization 02*, 2002. Course Notes, Tutorial 4.

[26] P. Sulatycke and K. Ghose. A fast multithreaded out-of-core visualization technique. In *PDP '99: Parallel and Distributed Processing*, pages 569–575, 1999.

[27] L. Xu and H.-W. Shen. Flow Web: a graph based user interface for 3D flow field exploration. In *Proceedings of IS&T/SPIE Visualization and Data 2010*, volume 7530, page 13, 2010.