Parallel Particle Advection and FTLE Computation for Time-Varying Flow Fields

Boonthanome Nouanesengsy*, Teng-Yok Lee*, Kewei Lu*, Han-Wei Shen*and Tom Peterka†
*Department of Computer Science and Engineering, The Ohio State University, Columbus, OH 43210
Email: {nouanese,leeten,luke,hwshen}@cse.ohio-state.edu
† Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL 60439
Email: tpeterka@mcs.anl.gov

Abstract—**Flow fields are an important product of scientific simulations. One popular flow visualization technique is particle advection, in which seeds are traced through the flow field. One use of these traces is to compute a powerful analysis tool called the Finite-Time Lyapunov Exponent (FTLE) field, but no existing particle tracing algorithms scale to the particle injection frequency required for high-resolution FTLE analysis. In this paper, a framework to trace the massive number of particles necessary for FTLE computation is presented. A new approach is explored, in which processes are divided into groups, and are responsible for mutually exclusive spans of time. This pipelining over time intervals reduces overall idle time of processes and decreases I/O overhead. Our parallel FTLE framework is capable of advecting hundreds of millions of particles at once, with performance scaling up to tens of thousands of processes.**

I. Introduction

Flow fields are a common and intensely studied class of data in many scientific, medical, and engineering fields. Scientists in such areas as combustion, climate modeling, and aeronautics all require analysis and insight into vector fields.

One of the most common and intuitive methods to visualize a vector field is to employ particle advection, where seeds are placed within the vector field, and their paths are recorded over a period of time. Given a set of particle seeds, their traces can help scientists understand the flow by clearly illuminating flow features, such as vortices and critical points. For time-varying vector fields, the trace of a particle over time forms a pathline. Various visualization and analysis algorithms require a large number of pathlines, including interactive flow visualization for large scale data [1][11], segments of flow fields [27], and detection of flow features [24].

A recent vector field analysis technique requiring the computation of pathlines is the Finite-Time Lyapunov Exponent (FTLE) [15]. Given a space-time location in a time-varying flow field, its FTLE value indicates how particles seeded around this location diverge after a finite number of time steps. The number of time steps used is dependent on the application. Multiple FTLEs can be generated from the same time-varying dataset to analyze how features evolve over time. Also, multiple FTLEs can be used to create animations of the FTLE field. Ever since Haller indicated that distinguished Lagrangian structures can be computed from FTLE values [15], the visualization community has utilized FTLEs to assist in various visualization tasks, including the placement of flowlines, [5] streaksurfaces [12], and visualization and analysis of Lagrangian structures [19][26].

The computation of multiple FTLEs require densely placed pathlines over both space and time. To compute one FTLE field for the entire time-varying vector field, ideally a pathline should be seeded from each spatial grid point at a certain time. In order to gain more insight about how the vector field changes over time, multiple FTLEs need to be computed, in which the seeds are placed at different times for each FTLE. Thus, advecting all of these particles is the main performance bottleneck in FTLE computation. In this paper, we explore a method for parallel computation of pathlines, with the main application being the generation of FTLEs in parallel.

While several algorithms have been proposed to accelerate the computation of FTLEs [4][13][19], the use of parallelism to accelerate FTLE computation has not been mentioned. Given the massive amount of particle advection required for large scale data, the use of supercomputers is the only reasonable choice to compute FTLEs within a practical time frame.

In this paper, we present an efficient algorithm for parallel time-varying particle advection, specifically for the requirements needed to compute multiple FTLEs for large scale vector fields, which require large numbers of particles distributed over both space and time. The main contribution of our algorithm, besides the conventional method of parallelizing over space, is to parallelize over time, creating a pipelined workflow structure, which allows particles in multiple time intervals to be advected in parallel.

Our algorithm first divides the processes into groups, and then assigns consecutive time intervals to the process groups in round-robin order. Each process group can load their time intervals as needed during run time, advect particles within the assigned time interval, and asynchronously communicate with other processes to send particles out for further advection. Since each process group works in parallel, the I/O time is overlapped with computation time, masking I/O latency, reducing idle time, and increasing efficiency.

The main contribution of this paper is an efficient algorithm to advect particles for large scale vector fields. While our main focus is to efficiently compute FTLE fields, such an algorithm can be used for other applications that also require a large number of particle traces, such as the advection of streaklines

SC12, November 10-16, 2012, Salt Lake City, Utah, USA
978-1-4673-0806-9/12/$31.00 ©2012 IEEE

and streaksurfaces, or precomputation of pathlines for interactive visualization. Likewise, other FTLE acceleration strategies can be easily combined with our particle advection algorithm.

In Section II, we cover the mathematical background behind particle advection and FTLE computation, and review related work in those fields. In Section III we discuss an overview of our parallel particle advection system. Section IV discusses the pipelined parallel advection model in detail. Timing results appear in Section V. In Section VI, we discuss limitations of our framework, and outline future work. We conclude with Section VII.

II. Background

This section reviews the related mathematical background for particle advection and FTLE computation. Applications and recent advances for parallel particle advection are discussed. Related research about FTLEs is also overviewed, including its applications, and progress into FTLE acceleration techniques.

A. Particle Advection

Given a time-varying 3D flow field $\mathbf{f} : \mathbb{R}^3 \times \mathbb{R} \to \mathbb{R}^3$, $\mathbf{f}(\mathbf{x}, t)$ defines the velocity at a point $\mathbf{x}$ in $\mathbb{R}^3$ at time t. By advecting a massless particle from $\mathbf{x}_0$ and t_0 along the local velocity $\mathbf{f}(\mathbf{x}, t)$, the trace of this particle indicates how the particle moves to other locations and time. The advection is equivalent to computing an integral curve $\mathbf{x}(t)$ such that

$$\mathbf{x}(t_0) = \mathbf{x}_0 \tag{1}$$

$$\frac{d}{dt}\mathbf{x}(t) = \mathbf{f}(\mathbf{x}(t), t) \tag{2}$$

namely, the tangent along the integral curve at a location $\mathbf{x}(t)$ and time t is equal to the vector velocity $\mathbf{f}(\mathbf{x}, t)$ at the same location and time. This integral curve is also called a *pathline*. If we are interested in the flow field at a time instance t_0, we can fix the time to t_0 when generating the integral curve. In this case, the derived integral curve is called a *streamline*, which can be expressed as $\frac{d}{dt}\mathbf{x}(t) = \mathbf{f}(\mathbf{x}(t), t_0)$.

Since it is often not possible to obtain a closed form solution, numerical methods are used to solve Equation 2, with vector data defined at integer time steps. Assuming that the step size is h, the continuous time instance t can be represented as $0, h, 2h, \ldots, ih, \ldots$ where $\mathbf{x}(ih) = \mathbf{x}(t)$. One of the most popular techniques to solve this equation is the fourth-order Runge-Kutta method, which is referred to as *RK4*. Given a point at location $\mathbf{x}(ih)$ and time step ih, the RK4 method computes the next particle position by looking up four additional steps via Euler methods, and then combines the four steps via Equation 3 as a more precise approximation for $\mathbf{x}((i+1)h)$:

$$\mathbf{x}((i+1)h) = \mathbf{x}(ih) + \tfrac{1}{6}\left(k_1 + 2k_2 + 2k_3 + k_4\right) \tag{3}$$

where

$$\begin{aligned}
k_1 &= hf(\mathbf{x}(ih), ih),\\
k_2 &= hf(\mathbf{x}(ih) + \tfrac{1}{2}k_1, ih + \tfrac{1}{2}h),\\
k_3 &= hf(\mathbf{x}(ih) + \tfrac{1}{2}k_2, ih + \tfrac{1}{2}h),\\
k_4 &= hf(\mathbf{x}(ih) + k_3, ih + h).
\end{aligned}$$

In this paper, we employ a fourth and fifth order embedded Runge-Kutta technique utilizing adaptive step size adjustment.

Since particle traces illustrate how the flow behaves across different regions over time, particle advection plays a significant role in flow visualization. A recent survey by McLoughlin *et al.* [20] provides a comprehensive overview on this topic. In addition to visualization, particle traces can be further analyzed to understand flow fields. The user can query for particle traces of interest by sketching [30] or explicit mathematical expressions [27][28], or statistics from pathlines can be used to detect flow features [18][24][29]. For these applications, it is desirable to densely place seeds in the domain in order to capture salient flow structures, thus requiring an efficient algorithm to advect a large number of particles.

In order to efficiently advect particles for very large datasets, several studies about generating streamlines for static flows on supercomputers have been presented. Pugmire *et al.* conducted a study using different strategies to parallelize streamline computation [25]. The impact of recent hardware advances, including multi-core architectures [7] and solid state disks [6], were also studied. Several algorithms have been proposed to distribute data blocks according to different criteria, including using spectral clustering to segment flow fields into blocks with minimal communication overhead between processes [8], or using non-convex quadratic programming to balance the distribution of block workloads [21].

In terms of generating pathlines in parallel, which is the focus of this paper, existing algorithms might advect particles within a limited space-time boundary [31], or load all time steps before advection [17], which may not be practical for extremely large data. For large scale time-varying data, having the flexibility of only needing to load a subset of the data to memory at once is important. The study conducted by Peterka *et al.* [23] illustrates that, when generating pathlines over space and time, the workload per process can vary because of the differences among the local flow field. As a result, the blocks of a time-varying vector field should be carefully distributed to processes in order to achieve a load-balanced computation.

B. Finite-Time Lyapunov Exponent

Here we briefly overview the definition and computation of the Finite-Time Lyapunov Exponent, or FTLE. Given a location $\mathbf{x}$ and a time t in a flow field, its FTLE value, denoted as $\phi_t^T(\mathbf{x})$, is a scalar that measures the separation between particle traces which are seeded spatially close to $\mathbf{x}$, begin at time t, and expire at time $t + T$. As introduced by Haller [15], the separation can be measured as the Jacobian of the *flow map*. For a flow field, its flow map $F_t^T(\mathbf{x}) : \mathbb{R}^3 \to \mathbb{R}^3$ returns the location of a particle at time $t + T$, where the

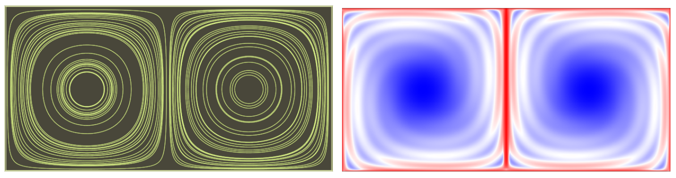

(a) (b)

Fig. 1. An example of a flow field and its corresponding FTLE field. (a). A set of particle traces of the vector field. (b). The corresponding FTLE field. The color blue denotes low FTLE values, while red denotes large FTLE values. The center contains a red vertical area of high FTLE values because particles seeded in that area will either flow to the left or right side, diverging greatly.

particle was originally located at point x and time t. If the seeding location is slightly shifted from $\mathbf{x}$ to $\mathbf{x} + \triangle\mathbf{x}$ at time t, the Jacobian of $F_t^T(\mathbf{x})$, denoted as $\nabla F_t^T(\mathbf{x})$, multiplied by the offset $\triangle\mathbf{x}$ indicates the coordinate offset at time $t+T$. Therefore, the square root of the maximal eigenvalue $\lambda_t^T(\mathbf{x})$ of $\nabla F_t^T(\mathbf{x})^\top \nabla F_t^T(\mathbf{x})$ indicates the maximum offset length if the seeding location is shifted by one unit away. Based on the maximum eigenvalue, the FTLE value is computed using Equation 4:

$$\phi_t^T(\mathbf{x}) = \frac{\log\sqrt{\lambda_t^T(\mathbf{x})}}{T} \tag{4}$$

An example of an FTLE field is shown in Figure 1. Particle traces are used to represent the input vector field, which contains two distinct regions of flow. The boundary of the two flow regions lies in the center, where FTLE values in that region are also high. This is because given a set of seeds placed in a small neighborhood in that region, particles will flow to completely different regions of the flow field, leading to high separation and high FTLE values.

The length of time particles are advected for an FTLE, T, depends on the application. In general, larger values of T will result in FTLEs with more refined features. Normally, the value of T will be less than the total number of timesteps available, which means that multiple FTLEs can be computed from one time-series dataset by using choosing different values of the start time t. Generating a sequence of FTLE fields can be useful for analyzing how flow features evolve over time. The standard method of computing a sequence of FTLEs is shown in Figure 2. For each FTLE, the length of time that particles are advected is constant, with the starting time of each FTLE offset by a fixed amount. This configuration results in particles being seeded in several different time steps, a feature which the pipelining technique described in Section IV takes advantage of.

In order to compute the highest resolution FTLE possible, a pathline should be seeded from each spatial grid point, the cost of which can vary from extremely time-consuming to totally impractical. Therefore, several algorithms have been proposed to accelerate the computation of FTLEs. One strategy is to approximate the flow map, which can be achieved by either reducing the seeding density over the spatial domain

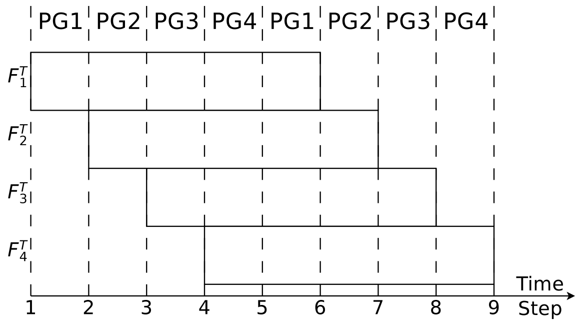


Fig. 2. Multiple flow maps with overlapping time spans. Here, four flow maps of time span $T = 5$ are shown. At the top, time intervals $[i, i+1]$, $i = 1 \ldots 8$, is assigned to 4 process groups (PG) in round-robin order.

[12][13][26], or combining flow maps with shorter time spans [4]. Compared to research on different FTLE approximation schemes, the use of parallel computing for FTLEs is a relatively new idea. One direction to efficiently compute FTLEs is using GPUs or APUs (Accelerated Processing Units) [10][14], which are confined to mid-size data sets because of limited memory size on GPUs and workstations. For large scale data, parallel computation of FTLEs will require the use of HPC platforms, which has been not addressed to our knowledge.

III. System Overview

Our parallel FTLE implementation is based on OSUFlow, a parallel particle advection library [23]. For the nearest neighbor communication required by particle advection, we employ the DIY library [22], modified to implement our parallel advection method described below. For all I/O tasks, the Block I/O Layer (BIL) library is utilized, which is designed to deliver high I/O performance when loading blocks of data across processes [16]. In the beginning, each process informs BIL of its block requests. BIL then reorganizes and merges the requests in such a way that each process will only request one contiguous read from the filesystem. Data blocks are then exchanged among processes in order to complete the original requests.

An overview of our parallel FTLE computation framework is shown in Figure 3. The framework can be divided into three main phases: initialization, particle advection, and FTLE computation. The initialization phase begins by taking the time-varying vector field, and decomposing it into blocks. All blocks are 4D space-time blocks, with extents in three spatial dimensions and the time dimension. To divide the data into blocks, first the entire time extent of the data set is divided into several different time windows. For each window of time, the spatial region is decomposed into several blocks that all share the same time extents as the time window. The set of all blocks within a certain time window is referred to as a *time interval*. The length of a time interval (the number of timesteps within a time interval) is a configurable parameter in our program.

An assignment of data blocks to processes, also called a partitioning, then needs to be calculated. The partitioning used is a major factor in the load balance of the computation. Since the amount of work for a block depends on the underlying flow field directions, it is difficult to determine the amount of load balance a partitioning will provide beforehand. Peterka *et al.* [23] illustrated that a block-cyclic round-robin scheme for time-varying pathline computation provided good results, so we choose to use round-robin as our partitioning scheme. Our results, detailed in Section V, support this claim.

Once those steps are completed, the seeds needed to compute the flow maps required by FTLEs are placed in the correct blocks. The parameters for the FTLE computation include the number of FTLEs to calculate, the start time for the flow map, how long particles are advected for each flow map, and the amount of subsampling for each dimension. From all these parameters, the locations of all necessary seeds can be computed and placed in the correct block.

Once all initialization steps are complete, the next phase is to compute the flow map by advecting pathlines. Earlier implementations of parallel pathline computation involved having all processes load one time interval at a time, advect all particles until they have terminated or left the time interval, then load the next time interval, continuing advection [23]. This continues until all particles have terminated. Since time intervals are loaded one at a time in chronological order, we refer to this method as the *time-serial method*.

In this paper, we explore the use of loading multiple time intervals at once, having particles advance through time intervals in a pipelined manner. All processes are evenly split into *process groups*. Process groups are responsible for loading particular time intervals, with a round-robin scheme used to assigned time intervals to process groups. When a process group is done with its current time interval, it loads its next assigned time interval. For each of its assigned time intervals, a process group is responsible for advecting particles until there are no more particles available for the time interval. As particles are advected, they are passed on to the next time interval if necessary. When particles cannot be advected any further, they are sent to the process which holds the flow map entry from which the particle originated from. We refer to this process as the *originating process*. This continues until all particles have been terminated or the last time interval has completed.

Since having multiple process groups advecting particles through their own time interval effectively creates a pipelined workflow, we refer to this technique as the *pipeline method*. Figure 2 illustrates how time intervals are assigned to process groups, and an example of how a mutli-FTLE computation is processed by the pipeline method is shown in Figure 4. Details of the pipelined advection method are further discussed in Section IV.

Once all particles have stopped advecting and have been sent back to their originating process, the advection phase is over, and the rest of the steps needed to finish FTLE calculation, such as flow map ghost cell exchange and Jacobian calculation, can proceed. Since our method can support multiple FTLEs, each individual FTLE field is assigned to

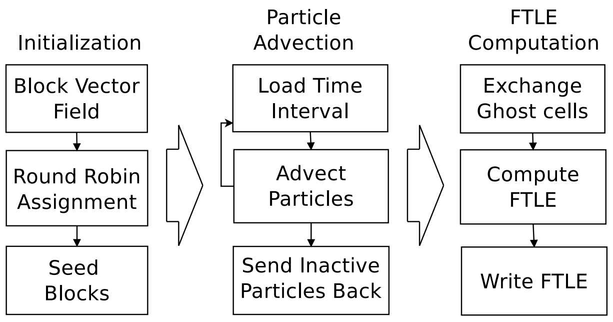


Fig. 3. An overview of our parallel FTLE framework

a process group, with the assignment again based on round-robin ordering. Each FTLE is then decomposed into blocks over all processes in the process group, with each process responsible for one block of the FTLE. Since one of the remaining computation steps involves computing the Jacobian, which is essentially a gradient of the flow map, one ghost cell layer is needed for each FTLE block. Thus, once the flow map is computed, ghost cell values need to be exchanged among all processes in the group. Once ghost cell values are exchanged, the rest of the FTLE computation can be performed in parallel without any further communication, following Equation 4. The resulting FTLE field is then written to disk in parallel using MPI I/O.

IV. Pipelined Parallel Particle Advection

A. *Pipelined Particle Advection*

In the time-serial approach, all processes work on the same time interval before advancing to the next. One problem with this time-serial method is that idle time from load imbalance hurts performance. If a process does not have any particles to advect, then it must sit idle until the time interval completes. In the worst case, every process except one will be idle while waiting for the last particle to finish. Another drawback to the time-serial method is that if particles are seeded throughout several time intervals, seeds that are not in the first time interval must wait until the computation arrives at their time interval.

One technique that can be used to overcome these drawbacks is to process multiple time intervals concurrently. If more than one time interval is available in memory at the same time, particles that are advected in one time interval can be immediately sent to the next time interval. The time intervals form a pipeline, which particles must traverse through chronologically as they advect through the flow field. When all particles have moved past the earliest time interval currently in memory, the next time interval that has not yet been processed is then loaded. This concept is similar to Chiueh and Ma's implementation of a pipelined time-varying volume renderer [9].

Pipelining across time intervals is accomplished by evenly dividing the available processes into groups, and having time intervals be processed by a group, instead of all available processes. Figure 4 illustrates the pipelined particle advection

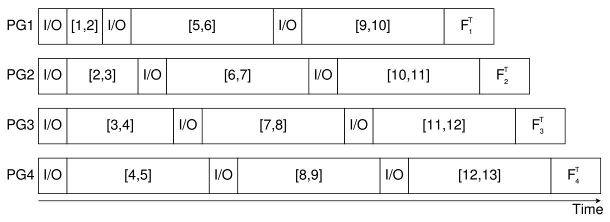


Fig. 4. Pipelined particle advection. At the beginning of the program, each process group (PG) loads its first assigned time interval, and begins to advect particles if possible. Time intervals are denoted by brackets indicating their start and end time. Each process group holds a time interval until all active particles have passed through the time interval. Once a process group is done with all its assigned time intervals, FTLEs assigned to that process group are then computed.

model. All process groups are assigned a certain number of time intervals. The time intervals a certain process group is responsible for will be processed in chronological order. At the beginning of the advection phase, all process groups will load their first time interval, and if any seeds lie within them, immediately begin to advect these particles. Whenever a particle is advected and reaches a block boundary, it is sent to the process containing the block necessary to continue advection. This process could be within the same process group, or in the process group responsible for the next time interval.

All particles within a time interval will be advected until they either exceed their maximum time, exit the data domain, or travel to the next time interval. Once all possible particles have moved past a time interval, the corresponding process group can move on and load the next time interval that has not yet been loaded. Eventually all time intervals will be loaded for particles to advect through.

Using a pipelined model provides two advantages compared to the time-serial method. The first advantage is that MPI collective operations have smaller communicators. In our parallel advection model, synchronization is only required when loading a new time interval from disk. Nevertheless, limiting this synchronization to a smaller number of processes should allow the pipelined model to scale better than the time-serial method. The second advantage when using pipelining is that the idle time of processes is reduced, since in a pipeline model there is a greater chance for processes to have particles to advect. The penalty for load imbalance within a time interval is also less severe, because imbalance is contained to only the processes in the group.

B. Asynchronous Particle Exchange

Throughout the advection phase, particles that need to continue traveling through the flow field are sent and received among processes, both within a process group and between groups. In order to obtain the best performance, the communication involved in transferring particles is designed to be as asynchronous as possible. Asynchronous communication allows processes to overlap compute with communication.

A particle is *active* if it is still advecting through the flow field. A particle becomes *inactive* if it exits the data domain, the maximum number of steps is reached, or the particle has reached its maximum time ($t + T$ in Equation 4).

Pseudocode of the algorithm used for computing particles in one time interval is shown in Figure 5. When the time interval begins, each process will begin advecting its local particles. When a particle becomes inactive, it is enqueued in a list of particles, called $ReturnList$, which holds particles that need to be sent back to their originating process. Otherwise, if a particle is still active, then it is enqueued in another list, called $ActiveList$, which holds particles that need to be sent to another process to continue advection. When the number of particles in $ActiveList$ exceeds a certain threshold, $P_{threshold}$, then all particles in the list are asynchronously sent to the necessary process to continue advection. When all local particles have been advected, the process checks if there are any more incoming active particles. If any are found, they are immediately advected. This continues until no more active particles can be found. Once it is determined that no more advection can be performed, other communication tasks are done, such as sending out particles stored in $ReturnList$, and receiving any inactive particles. All particle sends and receives are done asynchronously.

C. Time Interval Completion

A necessary operation of the pipeline method is determining when the task associated with a time interval is completed, at which time it is safe for the process group to advance to its next assigned time interval. A time interval is deemed completed when there is no further possibility that any particle will need to be advected within this time interval. In general, a time interval can only meet this criterion when there are currently no particles being advected within the time interval, and the time interval is chronologically the earliest time interval currently loaded in memory. If a time interval is not chronologically the earliest, then that means time intervals before it have not finished, and it is still possible for particles to be sent to this time interval.

One naive approach to determine whether a time interval is complete is to have all processes in the group count the number of active particles it currently has, then synchronize with a call to MPI_Allreduce to let all processes in the group know the current state. This can be periodically done until the time interval is determined to be completed. Performance will suffer, though, because this method will require constant polling and repeated synchronizations. A better approach is to use the method of Brunner *et al.* [2][3], along with some modifications to account for concurrent process groups.

This method for determining when a time interval is completed involves all processes in the group sending messages to a ‘root’ process, and vice versa. The root process will determine when the time interval is finished, and will have to inform every other process when it occurs. In order to avoid having only one process in the group repeatedly check and receive messages, which causes poor scaling and load imbalance, the communication overhead is spread out over all processes by using a binary-tree communication model. All communication within this binary tree is asynchronous.

Two quantities are maintained while advecting particles. The first number, called P_{total}, is the total number of active particles the time interval needs to compute. The second number, called P_{done}, is the number of particles that have either become inactive within this time interval, or have advected and proceeded to the next time interval. The current time interval is declared completed when $P_{total} = P_{done}$. The pseudocode describing how a time interval is determined to be done is shown in Figure 5.

Two sources of particles make up P_{total}. The first source is new particles seeded within this time interval, which we call P_{seed}. This occurs when one or more FTLEs begin in this time interval. From the FTLE computation parameters set by the user, this number can be easily computed, and requires no communication.

The second source is particles which were active in the previous time interval, and then were passed on to the current time interval. As particles are advected, each process keeps a count of the number of particles that were sent to the next time interval, called P_{next}. Whenever a time interval is declared done, the value of P_{next} needs to be communicated to the process group responsible for the next time interval, even if P_{next} is zero. In order to declare the current time interval completed and proceed to the next time interval, a message from the previous time group containing P_{next} must be received.

Instead of having all processes in the group send their value of P_{next} to the same destination, each process will send their value of P_{next} to their 'cousin' process, which is simply the process in the group responsible for the next time interval which has the same local group rank as the source process. In this way, each process in the group with the finished time interval will send one message, and each process in the next group will receive one message. Once the values of P_{next} are received, they are sent up the binary tree. A process will check their children for values of P_{next}, and when all values from its children have been received, only then will the process send its current total of P_{next} to its parent. The root knows the final value of P_{next} has been obtained when it has received a P_{next} value from each of the root's children processes, at which time the correct value of P_{total} can be calculated. Figure 6 illustrates how P_{next} is sent to the next process group, then is accumulated to the root.

The process of how P_{done} is acquired will now be described. Each process holds their own local value of P_{done}. During particle advection, whenever a particle becomes inactive or is sent to the next time interval, the local value of P_{done} is incremented. Eventually, all local values of P_{done} need to be summed up the binary tree and sent to the root. When not advecting particles, the process checks for any values of P_{done} from its children, then sends the current value of P_{done} to its parent. The local P_{done} is then reset to zero. If any more particles are advected after this point, the current value of P_{done} will eventually be sent again to the parent process.

```
 1: load blocks for time interval
 2: time_interval_done = False
 3: while time_interval_done is False do
 4:    while there are local particles do
 5:       for each local particle do
 6:          advect particle until block boundary, maximum
             number of steps, or time maximum is reached
 7:          if particle cannot be further advected then
 8:             particle is inactive, enqueue in $ReturnList$
 9:             increment $P_{done}$
10:          else
11:             particle is still active, enqueue in $ActiveList$
12:             if particle is in next time interval then
13:                increment $P_{next}$
14:                increment $P_{done}$
15:             end if
16:          end if
17:          if $ActiveList > P_{threshold}$ then
18:             send particles in $ActiveList$ to correct processes
19:          end if
20:       end for
21:       send particles in $ActiveList$ to correct processes
22:       check for any incoming active particles
23:    end while
24:    check child nodes for messages containing $P_{done}$
25:    if child messages are found then
26:       sum up received values with local $P_{done}$
27:    end if
28:    send $P_{done}$ to parent
29:    $P_{done} = 0$
30:    send particles in $ReturnList$ to correct processes
31:    check for any incoming inactive particles
32:    if have not received $P_{next}$ from previous cousin then
33:       check for $P_{next}$ from previous cousin
34:       if received $P_{next}$ from previous cousin then
35:          accumulate all values of $P_{next}$ at root
36:       end if
37:    else if i am root and $P_{total} == P_{done}$ then
38:       time_interval_done = True
39:       send done signal to children
40:    end if
41:    if received done signal from parent then
42:       time_interval_done = True
43:       send done signal to children
44:    end if
45: end while
46: send $P_{next}$ to cousin process in next group
47: unload blocks, goto Line 1
```

Fig. 5. The main loop for advecting particles in a time interval, including logic to determine when a time interval has finished. For simplicity, the logic to propagate P_{next} to the root is omitted.

As values of P_{done} propagate up the tree, eventually the values of P_{total} and P_{done} will be equal at the root of the binary tree. To ensure correctness, this equality check is not performed until the values of P_{next} are received from the previous group, and the complete value of P_{done} is obtained. Once it is determined that the time interval is complete, the root will send a done flag to all children, and the signal eventually propagates to all processes in the group. Once a process receives the done signal, it removes the blocks of the

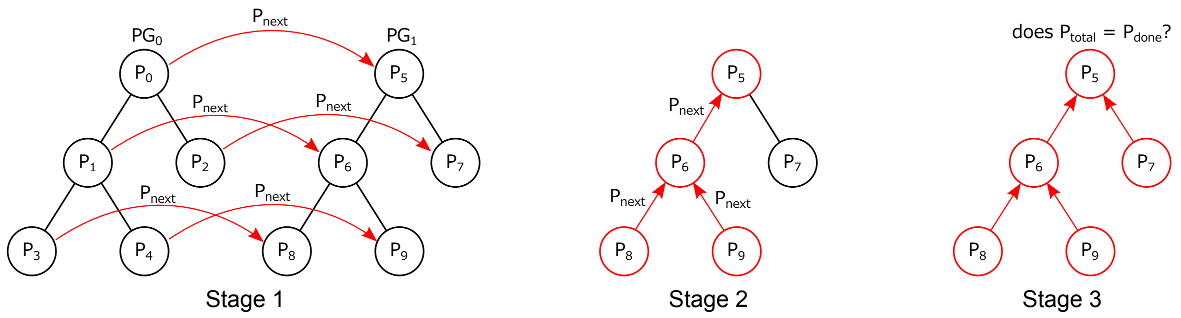


Fig. 6. An illustration of how P_{next}, the number of particles moving on to the next time interval, is communicated. Stage 1: When process group (PG) 0 is done, each process sends its value of P_{next} to their cousin in the next process group. Stage 2: Once P_6 receives P_{next} from its previous cousin and all of its children (P_8 and P_9), the total P_{next} is sent to its parent (P_5). Stage 3: When the root (P_5) receives P_{next} from its previous cousin and P_{next} from both children, the correct value of P_{total} is known and thus the root can start to check whether the time interval is done (whether P_{total} is equal to P_{done}).

current time interval from memory, and prepares to load the next time interval assigned to the group.

V. Results

We conducted several timing tests to study the performance of our parallel FTLE method. A total of four time-varying datasets were used. The *Plume* dataset is a simulation of the thermal downflow plumes on the surface of the sun. It contains 29 timesteps, each having a resolution of 126 x 126 x 512, which results in a total data size of 2.6 GB. The next dataset, *Ocean*, is the output from an eddy resolving simulation, with each timestep having a resolution of 3600 x 2400 x 40. With 36 timesteps total, the overall data size is 140 GB. The dataset *Isabel* is a hurricane simulation with each timestep having a resolution of 500 x 500 x 100. There are 48 total timesteps, for a total size of 13.4 GB. The last test dataset used is called *Smhagos*, and is a climate simulation over the Indian and Pacific Ocean. The number of timesteps used was 50, each with a resolution of 2699 x 599 x 27, for a total of 24 GB. Images of the FTLEs produced from these datasets are shown in Figure 7.

Over all timing tests, two operations dominated the running time: reading data from disk and particle advection. FTLE calculation steps performed after the flow map is complete, such as exchanging flow map ghost cells, computing the Jacobian, and writing the resulting FTLE field to disk, generally took less than one percent of the total running time. Therefore, we will not be listing those components in our results. Instead, we focus on the time required to read data from disk, which we refer to as I/O time, and advection time to generate the flow maps. The reported I/O times are the average I/O read time per process. Advection time includes communication of particles, advection computation (Runge-Kutta integration), other communication tasks, and possible idle time during the advection phase. Because of the asynchronous nature of our particle advection method, communication time and idle time cannot be easily recorded separately. The reported advection time is the average advection time per process. The total run time of the program is also detailed.

In our implementation, the sampling distance can be set separately for each spatial dimension. A sampling distance of one indicates that a seed is placed in every grid point in that dimension, while a value of two means every other grid point is sampled. For compactness, the sampling distance will be denoted as (i, j, k), where i, j, and k is the sampling distance in the x, y, and z dimension, respectively.

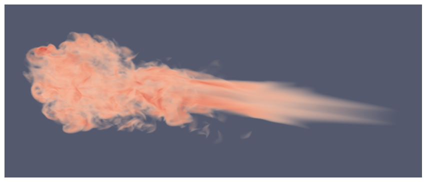
All tests were conducted on *Intrepid*, an IBM Blue Gene/P supercomputer situated at Argonne National Laboratory. *Intrepid* contains 40,960 nodes, each holding 4 cores, and utilizes the General Parallel File System (GPFS).

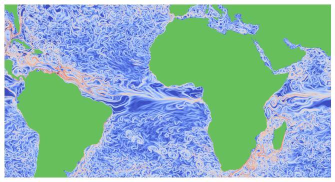
For all timing tests, the length of each time interval is set to 2. Longer time interval lengths require more memory, since more data is loaded at once per processes group. In *Intrepid*, each process has 500MB of memory available to them. Therefore, we use the minimum length of 2 timesteps per time interval to ensure that all timings tests did not run out of memory. Testing different lengths of time interval is a future work.

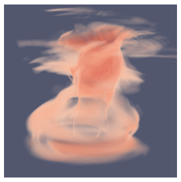
A. Varying Number of Blocks

An important parameter to consider when blocking data is the total number of blocks to create. As shown by Peterka *et al.* [23], when using round-robin partitioning, increasing the number of blocks will more evenly distribute load over all partitions. The downside to using more blocks is that as block size decreases, the surface area to volume ratio increases, resulting in more communication as particles will encounter block boundaries earlier and more often.

To test this parameter, we conducted runs using all four of our datasets, varying *bp*, the number of blocks per process. The different values of *bp* tested were 1, 2, 4, and 8. The number of process groups used was also varied with values of 1, 2, 4, and 8. For all tests, 1024 processes were used, and each FTLE computed spanned 6 timesteps. For *Plume*, 29 timesteps were used, with 23 total FTLEs computed. The sampling distance was set to (1, 1, 1), for a total of 186 million particles. The test involving *Ocean* used 32 timesteps, 26 FTLEs, and a sampling distance of (4, 4, 4). The total number of particles advected was 162 million. For *Isabel*, 48 timesteps were used and 42 FTLEs were calculated. A total of 65.6 million particles were produced with a sampling distance of (2, 2, 4). The *Smhagos*


(a) *Plume*


(b) *Ocean*


(c) *Isabel*

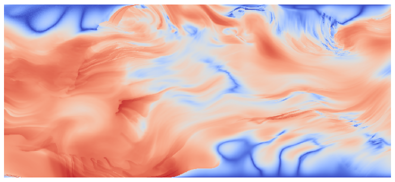
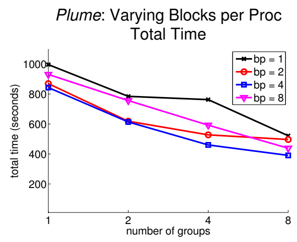
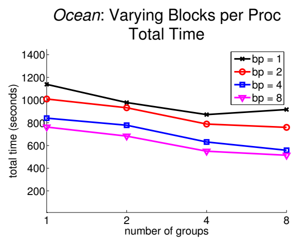

(d) *Smhagos*

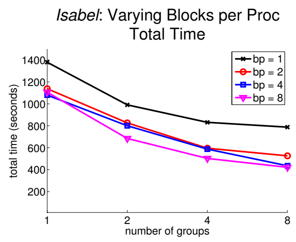
Fig. 7. Images of FTLE fields produced from our test datasets. Blue indicates low FTLE values, while red indicates high FTLE values.



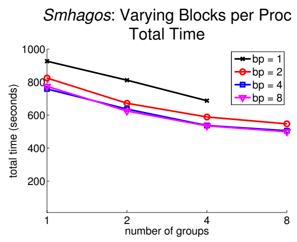
(a)



(b)



(c)



(d)

Fig. 8. The number of blocks per process is varied for all datasets. In general, a certain value of *bp* is preferred for each dataset. The *Plume* dataset performs best with *bp* = 4, while *bp* = 8 performs best with all other datasets.

test used 30 timesteps, 24 FTLEs, and a sampling distance of (4, 4, 2), for a total of 62 million particles.

Figure 8 shows the results of our tests. In general, performance increased as the number of blocks per process increased. For *Plume*, *bp* = 4 results in the best performance, while all other datasets prefer *bp* = 8. *Plume* results are different because the spatial size of *Plume* is smaller than those of other datasets. When *bp* = 8, the domain is divided into blocks which are too small, which increases communication time. The graphs in Figure 8 also illustrates a general trend of decreasing total time as the number of groups increase. This will be explored more in the next section.

B. Varying Number of Process Groups

The effectiveness of process groups is further investigated in Figure 9. Tests were conducted on *Isabel* and *Ocean* using 512, 1024, and 2048 processes, with the number of groups varying for each process count. The blocks per process is set to 8 for each run. The test parameters for *Isabel* are identical to the one used in Section V-A, except for blocks per process. For *Ocean*, 20 timesteps, 14 FTLEs, each with a time span of 6, are computed. Using a sampling distance of (4, 4, 4) results in 75.6 million particles.

The tests using *Isabel* show significant improvement in total time as the number of groups increases to 8. Both advection time and I/O time decrease, with I/O decreasing by a larger factor. When the number of groups becomes larger than 8, the total time begins to increase. This is due to advection time increasing, while on the other hand I/O time remains relatively stable.

Ocean tests exhibit similar behavior, with total time always decreasing as the number of groups is increased. Due to memory constraints, no more than eight groups could be used. For some processor counts, advection time increases going from four to eight groups, but this is offset by decreases in I/O time.

As shown most prominently in the *Isabel* tests, the advection time begins to increase once the number of groups is greater than a certain amount. This increase mainly stems from the fact that as the number of groups increases, more processes will stop participating in the last part of the advection phase. Consider the point in the advection phase when the number of time intervals which have not yet completed is less than the number of process groups. This means that some groups have processed all their assigned time intervals, and their only remaining task is to compute FTLE fields assigned to the group. Once FTLE computation is complete, though, they may become idle while other process groups perform particle advection. The number of processes not participating in the last stages of the advection phase will increase as the number of groups increase, thus advection time may suffer when a large number of groups is used, due to a low utilization of resources, as shown by Figure 9c.

Another notable impact as the number of groups is increased is the decrease in I/O time. For the *Isabel* run using 1024 processes, the I/O time has a speedup factor of 9 when going from one group to eight groups. I/O times decrease because as the number of groups increases, the number of time intervals assigned to any one group decreases. Therefore, the number of file open and close operations that any single process performs is reduced. Since the latency involved in file open and close is constant per file, the overall I/O time is reduced, assuming that the amount of bandwidth per process is constant. Furthermore, increasing the number of groups decreases the length of the pipeline in any one group, meaning there are fewer read stages. Because of this, we see significant drops in overall I/O time.

Overall, these tests show that using pipelining improves both

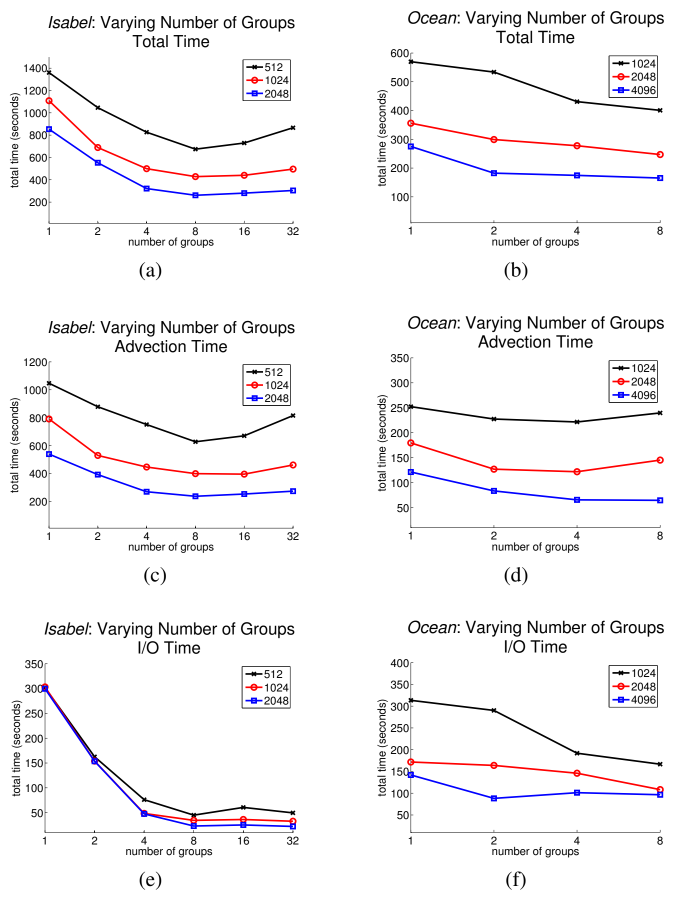


Fig. 9. Increasing number of groups are used for tests involving 512, 1024, and 2048 processes. Left column: results from *Isabel*. Right column: results from *Ocean*. For all process counts, the total time reduces when using up to eight groups. Performance degrades when too many groups are used.

advection time and I/O time. By using eight groups, total time is improved by a speedup factor of 3 for the *Isabel* tests, and 1.5 for the *Ocean* runs.

C. Strong Scaling

Strong scaling tests consisted of multiple FTLEs computed on each dataset using one and eight process groups, while scaling the total number of processes up to 16 K. When only one group is used, pipelining does not occur, and the algorithm essentially becomes the time-serial method. This way the scalability of the time-serial and pipelining method can be directly compared.

For *Plume*, the same parameters that were utilized in Section V-A are reused, with the exception being that a constant 4 blocks per process is set. The runs involving *Ocean* use 36 timesteps, with each FTLE spanning 6 timesteps, for a total of 30 FTLEs. The sampling distance is set at (3, 3, 4), which results in 288 million particles. The *Isabel* test uses 48 timesteps, and computes 42 FTLEs which each spans 6 timesteps, and uses 8 blocks per process. A sampling distance of (2, 2, 2) is used, for a total of 131 million particles. The tests using *Smhagos* uses 50 timesteps, and computes 44 FTLEs, each having a time span of 6. The sampling distance is (4, 4, 2), and the total number of particles is 62 million.

Results of the strong scaling tests are shown in Figure 10. The pipelined method employing eight groups consistently obtains higher performance versus the time-serial method. The charts illustrating parallel efficiency show that for *Ocean* and *Smhagos*, the pipelined method achieves much better parallel efficiency. For example, the *Ocean* tests achieve an efficiency of 0.45 using the pipelined method versus 0.19 using the time-serial method at 8 K processes. For *Plume* and *Isabel*, the parallel efficiency is similar, but the pipelined method obtains better overall timings.

D. Weak Scaling

Figure 11 shows the results obtained from a weak scaling study of each dataset. Similar to strong scaling, each test was performed using one group and eight groups. The number of particles and number of processes were doubled for each run. The number of particles is doubled by halving the current sampling distance in one dimension.

For *Plume*, the sampling distance at 512 processes is (2, 4, 4), which results in a sampling distance of (1, 1, 1) at 16 K processes. Tests for *Ocean* used a sampling distance of (16, 8, 8) at 1024 processes, which then becomes a sampling distance of (4, 4, 4) at 16 K processes. The sampling distance for *Isabel* at 512 process is (4, 8, 8), and at 16 K processes it grows to (2, 2, 2). For *Smhagos*, the initial sampling distance at 512 processes is (16, 8, 8), and is (4, 4, 2) at 16 K processes.

Results from the weak scaling test generally confirm what was previously indicated in the strong scaling tests. Using the pipelined method with eight groups always produced better results. Both *Ocean* and *Smhagos* results show diverging trends as the number of processes increases. The total time of the *Ocean* runs using the pipelined method actually decreases as the processes count increases, mainly due to the fact these runs are I/O bound.

VI. Limitations and Future Work

The main limitation when using the pipeline method with multiple process groups is that the memory requirements increase as more groups are used. Since each group works on a time interval, and all the data for the time interval is loaded at once, there must be enough memory among the processes in the group to hold all the data required.

Another limitation is that if there are no initial particles seeded in a group's first time interval, then processes must wait for particles from earlier time intervals to arrive. In general, the number of process groups should be less than or equal to the total number of FTLEs to prevent this issue.

Because of the fact that velocities must be interpolated between two timesteps, a timestep which is the last part of one time interval is the first component of the next time interval. This means that some timesteps will be loaded twice from disk. This occurs for both the time-serial and pipeline method. An optimization we are exploring is to load each timestep only

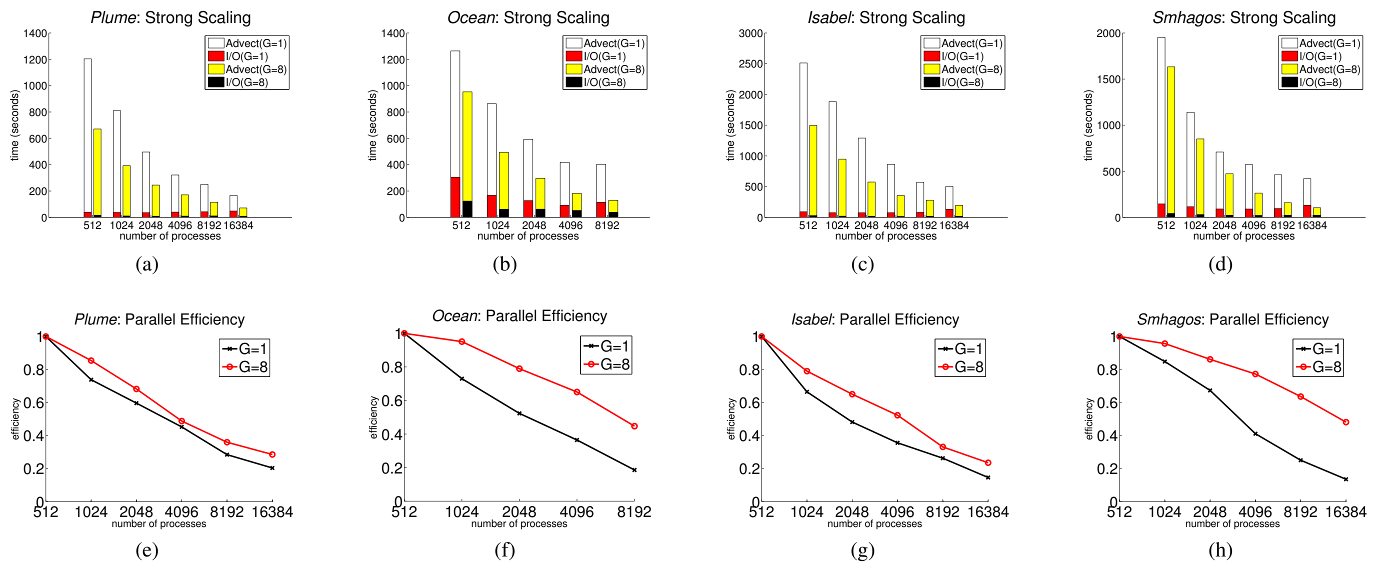


Fig. 10. Strong scaling results. The top row contains total time, separated into advection time and I/O time. The bottom row graphs parallel efficiency. Each column represents a different dataset. Using pipelining with eight process groups (G=8) consistently outperforms using only one group (G=1).

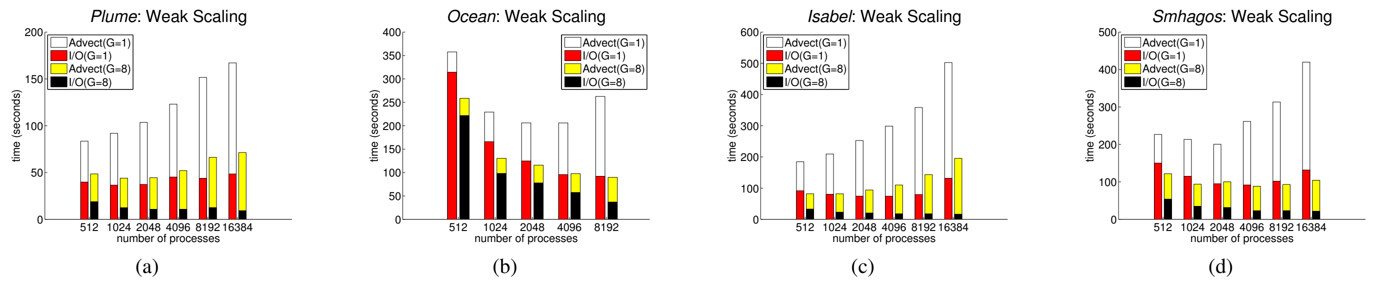


Fig. 11. Weak scaling results. Total time is shown, separated into advection time and I/O time. Each column represents a different dataset. Similar to the strong scaling results, using pipelining with eight process groups (G=8) consistently outperforms using only one group (G=1).

once, and have a process group send needed timesteps to other process groups.

For future work, we plan on making the advection more load balanced. Even though the pipelining method improves advection time and reduces overall idle time, a more informed partitioning could still enhance performance. Both preprocessing [21] and dynamic partitioning are possible avenues of research. Another potential improvement is to merge process groups together when a process group has completed all of its time intervals. In this way, better utilization of resources can be achieved.

VII. Conclusion

In this paper, we demonstrated a pipelined model for parallel particle advection. By separating processes into groups and advecting particles over multiple time intervals, advection and I/O performance are improved. Pipelining not only benefits FTLE computation, but can also be applicable to pathlines, streaklines, and streaksurfaces. Using this technique, we were able to advect hundreds of millions of particles, while scaling up to tens of thousands of processes.

Acknowledgment

We would like to thank the National Center for Atmospheric Research for providing the datasets *Isabel* and *Plume*, Mathew Maltrud of Los Alamos National Laboratory for the dataset *Ocean*, and Samson Hagos and Ruby Leung of Pacific Northwest National Laboratory for providing the dataset *Smhagos*. This research used resources of the Argonne Leadership Computing Facility at Argonne National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-06CH11357. This work was supported in part by NSF grant IIS-1017635, US Department of Energy DOE-SC0005036, Battelle Contract No. 137365, Los Alamos National Laboratory Contract No. 69552-001-08, and Department of Energy SciDAC grant DE-FC02-06ER25779, program manager Lucy Nowell.

References

[1] R. Bruckschen, F. Kuester, B. Hamann, and K. I. Joy. Real-time out-of-core visualization of particle traces. In *PVG '01: Proceedings of the IEEE Symposium on Parallel and Large-data Visualization and Graphics 2001*, pages 45–50, 2001.

[2] T. A. Brunner and P. S. Brantley. An efficient, robust, domain-decomposition algorithm for particle monte carlo. *Journal of Computational Physics*, 228(10):3882–3890, 2009.

[3] T. A. Brunner, T. J. Urbatsch, T. M. Evans, and N. A. Gentile. Comparison of four parallel algorithms for domain decomposed implicit monte carlo. *Journal of Computational Physics*, 212(2):527–539, 2006.

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

[5] K. Burger, P. Kondratieva, J. Kruger, and R. Westermann. Importance-driven particle techniques for flow visualization. In *PacificVis '08: Proceedings of the IEEE Pacific Visualization Symposium 2008*, pages 71 –78, 2008.

[6] D. Camp, H. Childs, A. Chourasia, C. Garth, and K. Joy. Evaluating the benefits of an extended memory hierarchy for parallel streamline algorithms. In *LDAV '11: IEEE Symposium on Large Data Analysis and Visualization 2011*, pages 57 –64, 2011.

[7] D. Camp, C. Garth, H. Childs, D. Pugmire, and K. Joy. Streamline integration using MPI-hybrid parallelism on large Multi-Core architecture. *IEEE Transactions on Visualization and Computer Graphics*, 17:1702–1713, 2011.

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

[9] T.-C. Chiueh and K.-L. Ma. A parallel pipelined renderer for time-varying volume data. In *I-SPAN '97: Proceedings of the Third International Symposium on Parallel Architectures, Algorithms, and Networks 1997*, pages 9 –15, 1997.

[10] 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.

[11] D. Ellsworth, B. Green, and P. Moran. Interactive terascale particle visualization. In *VIS '04: Proceedings of the IEEE Conference on Visualization 2004*, pages 353–360, 2004.

[12] F. Ferstl, K. Burger, H. Theisel, and R. Westermann. Interactive separating streak surfaces. *IEEE Transactions on Visualization and Computer Graphics*, 16(6):1569–1577, 2010.

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

[14] C. Garth, G.-S. Li, X. Tricoche, C. D. Hansen, and H. Hagen. Visualization of coherent structures in transient 2d flows. In *TopoInVis '07: Proceedings of the Workshop on Topology-Based Methods in Visualization 2007*, pages 1–13, 2007.

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

[16] W. Kendall, M. Glatter, J. Huang, T. Peterka, R. Latham, and R. Ross. Terascale data organization for discovering multivariate climatic trends. In *SC '09: Proceedings of the ACM/IEEE Conference on Supercomputing 2009*, pages 15:1–15:12, 2009.

[17] W. Kendall, J. Wang, M. Allen, T. Peterka, J. Huang, and D. Erickson. Simplified parallel domain traversal. In *SC '11: Proceedings of the ACM/IEEE Conference on Supercomputing 2011*, pages 10:1–10:11, 2011.

[18] A. Lez, A. Zajic, K. Matkovic, A. Pobitzer, M. Mayer, and H. Hauser. Interactive exploration and analysis of pathlines in flow data. In *WSCG '11: Proceedings of International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision 2011*, pages 17–24, 2011.

[19] D. Lipinski and K. Mohseni. A ridge tracking algorithm and error estimate for efficient computation of lagrangian coherent structures. *Chaos*, 20:017504, 2010.

[20] T. McLoughlin, R. S. Laramee, R. Peikert, F. H. Post, and M. Chen. Over two decades of integration-based, geometric flow visualization. *Computer Graphics Forum*, 29(6):1807–1829, 2010.

[21] 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.

[22] T. Peterka, R. Ross, W. Kendall, A. Gyulassy, V. Pascucci, H.-W. Shen, T.-Y. Lee, and A. Chaudhuri. Scalable parallel building blocks for custom data analysis. In *LDAV '11: IEEE Symposium on Large Data Analysis and Visualization 2011*, pages 105–112, 2011.

[23] T. Peterka, R. Ross, B. Nouanesengsey, T.-Y. Lee, H.-W. Shen, W. Kendall, and J. Huang. A study of parallel particle tracing for steady-state and time-varying flow fields. In *IPDPS '11: Proceedings of IEEE International Parallel & Distributed Processing Symposium 2011*, pages 580–591, 2011.

[24] A. Pobitzer, A. Lez, K. Matkovic, and H. Hauser. A statistics-based dimension reduction of the space of path line attributes for interactive visual flow analysis. In *PacificVis '12: Proceedings of the IEEE Pacific Visualization Symposium 2012*, pages 113–120, 2012.

[25] D. Pugmire, H. Childs, C. Garth, S. Ahern, and G. H. Weber. Scalable computation of streamlines on very large datasets. In *SC '09: Proceedings of the ACM/IEEE Conference on Supercomputing 2009*, pages 16:1–16:12, 2009.

[26] F. Sadlo, A. Rigazzi, and R. Peikert. Time-dependent visualization of lagrangian coherent structures by grid advection. In *Topological Methods in Data Analysis and Visualization*, pages 151–165, 2011.

[27] T. Salzbrunn, C. Garth, G. Scheuermann, and J. Meyer. Pathline predicates and unsteady flow structures. *Visual Computer*, 24(12):1039–1051, 2008.

[28] T. Salzbrunn and G. Scheuermann. Streamline predicates. *IEEE Transactions on Visualization and Computer Graphics*, 12(6):1601 – 1612, 2006.

[29] K. Shi, H. Theisel, H. Hauser, T. Weinkauf, K. Matkovic, H.-C. Hege, and H.-P. Seidel. Path line attributes – an information visualization approach to analyzing the dynamic behavior of 3d time-dependent flow fields. In *TopoInVis '09: Proceedings of the Workshop on Topology-Based Methods in Visualization 2009*, pages 75–88, 2009.

[30] J. Wei, C. Wang, H. Yu, and K.-L. Ma. A sketch-based interface for classifying and visualizing vector fields. In *PacificVis '10: Proceedings of the IEEE Pacific Visualization Symposium 2010*, pages 129–136, 2010.

[31] H. Yu, C. Wang, and K. L. Ma. Parallel hierarchical visualization of large time-varying 3D vector fields. In *SC '07: Proceedings of the ACM/IEEE Conference on Supercomputing 2007*, pages 24:1–24:12, 2007.