Particle Advection Workloads: Performance Characteristics and Optimizations

Abhishek Yenpure[1]

[1]University of Oregon, USA

Abstract
Flow visualization is an important approach for understanding fluid dynamics simulations. This survey focuses on flow visualization algorithms that use “particle advection,” a process that displaces particles in a flow field. The performance of these algorithms can vary greatly based on a variety of factors including workload, solver type, underlying mesh type, and optimizations employed. That said, the relationship between these factors and actual execution time is often not well understood. In response, this survey aims to illuminate performance aspects. It considers a decision-making workflow for assessing whether or not a particle advection workload requires optimized approaches to complete within a time bound. This workflow requires considering workload properties, cost models, and possible optimizations, and each of these topics is surveyed. Further, special attention is paid to parallelism and especially parallelism on supercomputers, including portable performance. Overall, the survey identifies three key limitations in realizing the decision-making workflow: in cost estimation, in expected performance increases from using GPUs, and in expected performance increases from using distributed-memory parallel techniques. Finally, the survey contributes new ideas for how particle advection components relate and for translating particle advection workloads to a cost formulation, also contributes some nascent preliminary work addressing each of the three limitations.

1. Introduction

Flow visualization is a technique for understanding flow patterns and movement of fluids. It is performed both experimentally and computationally, with computational approaches falling within the field of scientific visualization. It is used by experts from various fields to study phenomena like ocean movements, aerodynamics, and electromagnetics. In the context of scientific visualization, almost all flow visualization techniques utilize the same operation — placing a massless particle at a seed location, displacing that particle according to the vector field to form a trajectory, and using that trajectory to create a renderable output. Many of these flow visualization techniques consider large numbers of particles and thus can be very computationally expensive. However, the exact amount of computational work varies from case to case.

The main focus of this study is the computational costs for flow visualization, and it considers these costs from two perspectives: (1) reasoning about how much work a flow visualization technique will require and (2) reasoning about how to execute this work as quickly as possible. We envision the results of this survey can help inform a workflow for making decisions about which optimizations to employ for particle advection problems. The workflow is shown via a flow chart in Figure 1, and works in three steps:

- In the first step, the desired workload is analyzed to see how many operations need to be performed.
- In the second step, the analysis from the first step is used to estimate the execution time costs to execute the algorithm.
- In the third step, the estimated costs from the second step are compared to user requirements. If the estimated costs are within the user’s budget, then no optimizations are necessary and the workload can be executed as is. If not, then optimizations should be employed.

With respect to optimizations, there are two main types. One type of optimization is algorithmic, i.e., to do less work. Another type of optimization is parallelization, i.e., to use more resources to do the same work. In many cases, these optimizations can be used in conjunction. Utilizing available parallelism can usually be considered the easier option. However, depending on the size of the workload and the data, further decisions are needed regarding utilizing shared-memory parallelism or distributed-memory parallelism.

Additionally, this survey considers parallelism on supercomputers. As such, the parallelism study is split into two parts, shared-memory techniques and distributed-memory techniques, and particularly focuses on the latter. The discussion of distributed-memory techniques is intended to serve as an exhaustive survey of the works in this area to date, and also go deeper in contrasting these works than any previous survey. Fi-

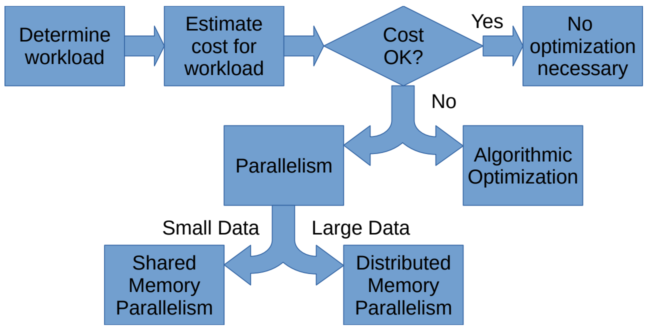
Determine workload
Estimate cost for workload
Cost OK?
Yes
No optimization necessary
No
Parallelism
Algorithmic Optimization
Small Data
Large Data
Shared Memory Parallelism
Distributed Memory Parallelism

Figure 1: *A flow chart to determine the potential optimizations to be applied to a flow visualization algorithm.*

nally, the evolving nature of supercomputer hardware has made portable performance an important issue. To that end, this study analyzes portable performance results to date for scientific visualization.

In all, the organization of this document is:

- Section 2: background on flow visualization
- Section 3: reasoning about how much work a flow visualization technique will take
- Section 4: reducing the amount of work via algorithmic optimizations
- Section 5: shared memory parallelization
- Section 6: distributed memory parallelization
- Section 7: portable performance
- Section 8: conclusion

During the course of this survey, we identified three areas where existing studies do not sufficiently provide answers we need to realize the aforementioned workflow. We highlight these areas as ***Limitation 1***, ***Limitation 2***, and ***Limitation 3*** in the manuscript. In each case, we also augment current understanding with some nascent preliminary work that reveals the extent of the limitation. Briefly, the limitations are as follows:

- **Limitation 1**: Estimating execution time for arbitrary workloads is difficult, and may require a formal cost model. For our preliminary work, we make a cost model based on the number of operations performed and briefly evaluate its efficacy. We find that merely counting operations is not sufficient to capture diverse workloads, and likely that considerations for memory accesses are needed.
- **Limitation 2**: For shared-memory parallelism, our community has a poor understanding of expected speedups over diverse GPUs and diverse workloads. For our preliminary work, we complement existing studies that use powerful GPUs on difficult workloads with a mini-study using less powerful GPUs and less intensive workloads and show that the achieved speedups differ dramatically.
- **Limitation 3**: For distributed-memory parallelism, our community has a poor understanding of scaling properties for our algorithms. For our preliminary work, we re-analyze previous results to inform parallel efficiency, although these results are for weak scaling, and our intended workflow would most benefit from a study focused on strong scaling.

2. Background

This section provides background on flow visualization, organized into three subsections. Section 2.1 introduces an organization for the components of a flow visualization algorithm. These components fall into two broad groupings. The first grouping is made up of components that are general-purpose building blocks that can be utilized by any flow visualization algorithm. These components are described in Section 2.2. The second grouping is made up of components that are individual to a given flow visualization algorithm. Section 2.3 informs the nature of these components by surveying some representative flow visualization algorithms, with a focus on how they realize the various components from the second grouping.

2.1. Organization for the Components of a Flow Visualization System

This subsection introduces an organization for the components of a flow visualization algorithm. This organization considers three levels of granularity. The “top level” of our organization is the coarsest granularity, and it contains three components:

- ***Seed Particles:*** defines the initial placement of particles.
- ***Advance Particles:*** defines how the particles are displaced and analyzed.
- ***Construct Output:*** constructs the final output of the flow visualization algorithm, which may be a renderable form, something quantitative in nature, etc.

These operations happen in a sequence, starting with the placement of the seed particles. The particles are then advanced by the flow visualization system. Finally, the information from the particle advancements are used to construct the visual output for the users.

The “middle level” of our organization considers the process of advancing a single particle. This is an expansion of the second component of the top-level, “Advance Particles.” It is again divided into three parts:

- ***Particle Advection Step:*** defines how a particle advances to the next position.
- ***Analyze Step:*** defines the analysis that is required after every step of a particle.
- ***Check for Termination:*** defines when a particle should be terminated.

The process of advancing a particle involves taking many steps, with each step displacing the particle from its current location to a new location. Such steps are referred to as particle advection steps. For every particle being advanced, the particle is first displaced from its current location for a single step. Some flow visualization algorithms require special analysis of the displacement’s outcome. This analysis could be as simple as storing the particle’s new location in memory or could involve more computation. Further, flow visualization algorithms define specific

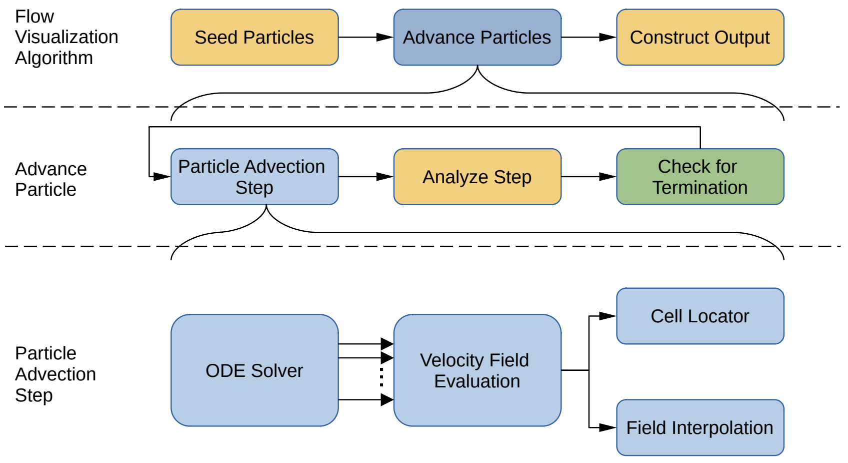

Figure 2: *Organization of the components for a particle advection-based flow visualization algorithm. The components are categorized in two ways. The first categorization is based on the hierarchy at which the components occur. The components are arranged in three rows in decreasing levels of granularity from top to bottom. In other words, the components at the bottom are building blocks for the components at higher levels. The top row shows the components which encapsulate a flow visualization algorithm. The middle row shows components that define the movement and analysis of a particle. The loop in this middle level indicates its components are executed repeatedly until the particle is terminated. Note that, in the figure, the top row advances many particles, while the middle row is dedicated to advancing a single particle, i.e., a flow visualization algorithm would advance each of its particles by repeatedly using the components in the middle column. Finally, the bottom row shows components that define a single step of advection. The four arrows with the ellipsis from ODE solver to velocity field evaluation are meant to indicate that an ODE solver needs to evaluate the velocity field multiple times. Each velocity field evaluation takes as input a spatial location and possibly a time, and returns the velocity at the corresponding location (and time). A frequently used ODE solver, Runge-Kutta 4, requires four such velocity field evaluations. Further, each velocity field evaluation requires first locating which cell in the mesh contains the desired spatial location and then interpolating the velocity field to the desired location. The second categorization is based on the role the components play. The components are organized by different colors. The colors distinguish between common components and components that are custom for the flow visualization algorithms — the blue components are the key components of particle advection which are common to all algorithms, and the yellow components are the components that are custom to an algorithm. The "Check for Termination" component is colored in green (blue + yellow) because it can fall into either group. There are several common termination criteria that are used repeatedly across flow visualization techniques, but some flow visualization techniques have custom criteria.*

criteria for when to terminate a particle. After analyzing the particle's displacement, a special component checks if the particle meets the termination criteria. Further, if the particle is not terminated, then these three activities are repeated in a loop until the termination criteria is reached.

Finally, the "bottom level" of our organization is the finest granularity and considers the process of completing a single step for a particle. This level has four components:

- ***ODE solver*** (ordinary differential equation solver): calculates the particle's next position using velocity field evaluations.
- ***Velocity Field Evaluation:*** calculates the value of the velocity field at a specific location.
- ***Cell Location:*** locates the cell that contains some location.
- ***Field Interpolation:*** calculates velocity field at a specific location via interpolation of surrounding velocity values.

These four components relate as follows. For every particle step, the particle is displaced to a new location determined by an ODE solver. This ODE solver requires the velocity of the particle for the displacement. The velocity evaluation is performed using two components 1) a cell locator to identify the cell containing a location, and 2) a field interpolator that calculates the velocity using the cell's information.

This organization is also described in Figure 2.

2.2. Building Blocks for a Flow Visualization Algorithm

This section describes the components from our organization that serve as general-purpose building blocks for any flow visualization algorithm.

2.2.1. Particle Advection Step

The trajectory a particle follows can be defined using an ordinary differential equation (ODE):

$$\begin{aligned} s'(t) &= v(t, s(t)) \\ s(t_0) &= s_0 \end{aligned} \tag{1}$$

where $s(t)$ denotes the position of the particle at time t, $s_0 \in \mathbb{R}^d$ is the position at initial time t_0, v is a function that returns the velocity for position s at time t such that $v : [t_0, \infty) \times \mathbb{R}^d \mapsto \mathbb{R}^d$, and $s'(t)$ is the derivative of the particle trajectory at time t. An advection step displaces a particle according to the vector field by approximating a solution to the ODE. If a particle is located at position P_i at time T_i, then an advection step displaces the particle to a new position, $P_i + \Delta p$, and a new time, $T_i + \Delta t$. Three operations are required for every particle advection step:

1. Locating the cell or cells for the locations required for the step.
2. Interpolating the velocity information at the locations required for the step.
3. Solving for the next position of the particle.

Algorithm 1 presents a simplified view of this routine. The first two operations are parts of the "vector field evaluation" component discussed earlier. The rest of this section describes each of these subcomponents of particle advection.

Algorithm 1 Algorithm for advancing a particle until termination. Lines inside the while loop are computations for a single step. K is the number of velocity evaluations required by the ODE solver.

```
while $!ShouldTerminate()$ do
    $vel \leftarrow \{\}$
    for i = 1 to K do
        $loc_i \leftarrow ODESolver.GetLocation(i)$
        $cell_i \leftarrow CellLocator.FindCell(loc_i)$
        $vel_i \leftarrow FieldInterpolator.Interpolate(cell_i, loc_i)$
        $vel \leftarrow vel + \{vel_i\}$
    end for
    $pos \leftarrow ODESolver.Solve(pos, vel)$
end while
```

2.2.1.1. ODE Solver: Solutions to ordinary differential equations can be approximated by using "ODE Solvers."

In the context of particle advection, ODE solvers are used to calculate the next position of the particle. This process is continued successively, i.e., each time a new particle position is calculated, that position becomes the input to the next step, and the result of each of these steps is an approximation of the particle's trajectory. There are many different solvers, each of which perform this approximation in different ways. These solvers have trade-offs in accuracy, computational cost, and suitability, and users choose a solver based on these trade-offs. The simplest scheme to perform this approximation, known as the Euler method, is shown in Equation 2.

$$P_{i+1} = P_i + \Delta t \times v(t_i,\ P_i) \tag{2}$$

In this equation, P_i is the particle's current position at time t_i, and P_{i+1} is the particle's next position at time $t_i + \Delta t$. The term $v(t_i,\ P_i)$ represents the velocity at position P_i. Finally, Δt represents the duration of the displacement with the evaluated velocity value.

Since the displacement uses only the velocity at the particle's current position and time, displacing the particle for a longer duration could potentially accumulate significant error in the particle's trajectory over time. One solution to this problem is to displace the particle for a smaller duration (e.g., smaller values for 'Δt' in Equation 2), while another is to use a higher-order integration method. The most commonly used integration method in flow visualization is the fourth-order Runge-Kutta method, which is presented in Equation 3.

$$\begin{aligned} P_{i+1} &= P_i + \frac{\Delta t}{6} \times (k_1 + 2k_2 + 2k_3 + k_4) \\ k_1 &= v(t_i,\ P_i) \\ k_2 &= v(t_i + \frac{\Delta t}{2},\ P_i + \frac{\Delta t}{2} \times k_1) \\ k_3 &= v(t_i + \frac{\Delta t}{2},\ P_i + \frac{\Delta t}{2} \times k_2) \\ k_4 &= v(t_i + \Delta t,\ P_i + \Delta t \times k_3) \end{aligned} \tag{3}$$

While the Runge-Kutta method requires more computations to calculate a particle's next position, it is able to preserve accuracy over longer durations.

Consider an example of a particle being advected with for a duration of $T = 1$ with $\Delta t = 0.01$ using an RK4 solver. In this case, the solver will need to complete $1/0.01 = 100$ steps. Over the entire advection, the error accumulated by the RK4 solver is of the order $O(\Delta t^4)$. In the case of the example, the accumulated error is $O(1e^{-8})$. Since the particle completes 100 steps, the RK4 solver requires 400 velocity evaluations. Over the entire advection, the error accumulated by the Euler solver is of the order $O(\Delta t)$. To attain the same accuracy as the RK4 solver, the Euler solver will need Δt to be $1e^{-8}$, requiring $1e^8$ velocity evaluations, to advect for the same duration. In comparison, the Euler solver requires $1e^8/400$, i.e., $250,000\times$ as many velocity evaluations. Hence, when considering both performance and accuracy, the RK4 solver provides a better option.

Euler and Runge-Kutta are just two of many numerical techniques for approximating ordinary differential equations. Other notable examples used for particle advection are Dormand Prince [DP80], Adams Bashforth [Gol12], and Leapfrog [Ske93].

2.2.1.2. Vector Field Evaluation: Simulations operate by discretizing their problem domains into "meshes" made up of many elements. The nature of these meshes varies widely across simulation code, including many different types of elements and organizations of these elements. Simulations typically define fields on their meshes, and most flow visualization algorithms operate by using the velocity vector field defined on the mesh. In this discussion, meshes are classified into two broad categories: ***structured*** and ***unstructured***. Structured meshes exhibit regularity in terms of the elements' shape and organization, which simplifies the specification of their topology. Examples of structured meshes are ***uniform***, ***rectilinear***, and ***curvilinear*** grids. Unstructured meshes do not exhibit regularity and need the topology of every element to be specified explicitly. For particle advection, a simulation's mesh type can significantly impact on the performance of the vector field evaluation operation.

All ODE solvers require evaluating the velocity field at a series of locations, $L_1, L_2, \ldots L_N$. Evaluating velocity at each location L_i is a two-step process. The first step is to determine which cell C_i contains L_i, which we refer to the "containing cell." The second step is to evaluate the velocity value at L_i using the velocity field information from C_i. Two components are used to execute these two tasks: a ***cell locator*** to perform the first and a ***field interpolator*** to perform the second. The remainder of this subsection further describes these two components.

Cell Locator: Cell locators are data structures that accelerate repeated searches for containing cells. The cost of this operation varies based on mesh type, in terms of both implementation and execution time. Finding the containing cell C_i for a location L_i is straightforward in uniform and rectilinear grids. In uniform grids, a simple operation with the location L_i and the grid's properties (spacing, resolution) can determine the containing cell. In rectilinear grids, a fast search can be applied for L_i's coordinates in different directions to determine the containing cell. Hence, uniform or rectilinear grids require only simple cell locator approaches. Curvilinear and unstructured grids, however, require sophisticated approaches. In particular, given that some meshes contain upwards of a billion elements, it is not practical to sequentially search each cell to see if it contains a location. A simple approach is to use spatial data structures like a k-d tree or an octree. That said, there are many additional techniques, many of which yield faster execution times. These techniques are reviewed in Section 4.1.

Field Interpolation: After finding a containing cell for a location requested by the ODE solver, the next step is to evaluate the velocity at that location. This operation requires gathering all the velocity values associated with the cell and then interpolating the value at the required location. The complexity of this operation is based on the type of the mesh and shape of the cell. For a uniform or rectilinear grid where elements are hexahedral, trilinear interpolation is a common way for calculating velocities. However, for other cell shapes, the approach for interpolation can vary. The operation may involves computing the Berycentric coordinates of a location within the cell and then interpolating the actual values. This process can be computationally intensive in the case of higher-order meshes.

2.2.2. Check for Termination

This component determines when a particle should be terminated. The two most common termination criteria for flow visualization techniques are:

1. The particle advects for some specific duration.
2. The particle advects for some specific distance.

Additional termination criteria are added based on the analysis requirements of the flow visualization technique, e.g., the number of orbits completed by a particle for a Poincaré plot (see Section 2.3.4 for more information). Further, to improve the efficiency of the flow visualization algorithm, a particle can be terminated if it gets stuck in a critical region. Finally, most implementations in practice place a bound on the maximum number of steps. For example, a particle in a region with slow-moving velocity may require too many steps to travel a specific distance.

2.3. Representative Techniques

This section reviews some flow visualization techniques that use particle advection. The discussion of each technique is organized

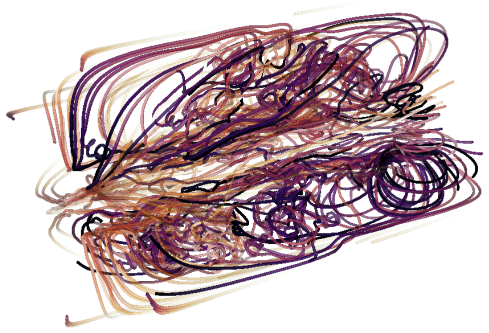

Figure 3: *Streamlines rendered over a slice of the jet plume data. The jet plume is a simulation of a jet of high-velocity fluid moving into a medium at rest, created using the Gerris Flow Solver [Pop03].*

into two parts. The first part is a general description of the techniques. The second part considers the four components of our organization that are specified by a flow visualization technique:

1. Seed Particles: The method used for seeding particles.
2. Analyze Step: Choice of analysis for a step of a particle.
3. Check for Termination: The termination criteria for a particle.
4. Construct Output: The method for construction of the flow visualization output.

The term “seeding” in particle advection refers to the initial number of particles and their placement for flow visualization algorithms. The number and placement of particles can have a significant impact on the quality of the analysis. Additionally, the number of steps completed by each particle can significantly impact the flow visualization algorithm's performance. The usual ranges for these parameters are shown in Table 1.

2.3.1. Streamline and Pathline

Streamlines and pathlines are related techniques that differ based on the type of vector data they handle. This description begins by describing a streamline and then discusses pathlines by comparing its difference with streamlines.

A streamline plots the entire trajectory a particle travels. Streamlines use line segments (or tubes) to connect the series of locations a particle travels to as it is advected. Streamlines operate on steady state vector fields, i.e., vector fields that do not change/evolve with time. An example of streamlines visualizing a flow is presented in Figure 3. Pathlines are the extension for streamlines for visualizing unsteady vector fields, i.e., vector fields that change with time. In terms of practical implementation, pathlines often require temporal interpolations. In a typical scenario, only “time slices” of data are stored to disk, and pathline calculations require evaluating velocities at times that occur between saved time slices.

Seeding Strategy	***Sparse***: particles are distributed sparsely in the specified region of the data.	***Packed***: particles are distributed densely in the specified region of the data.	***Seeding Curves***: particles are placed along a curve defined by the seeding curve.
Number of Seeds	***Small***: ≤1/1K cells	***Medium***: 1/100 cells	***Large***: ≥1/cell
Number of Steps	***Small***: ≤100	***Medium***: 1K	***Large***: ≥10K

Table 1: *Parameter classification for seeding strategy, the number of seeds, and the number of steps for flow visualization algorithms*

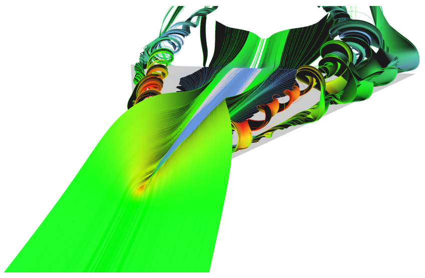

Figure 4: *Streamsurface rendered over a vector field. The surface is split by turbulence and vortices can be observed towards the end [Fis13].*

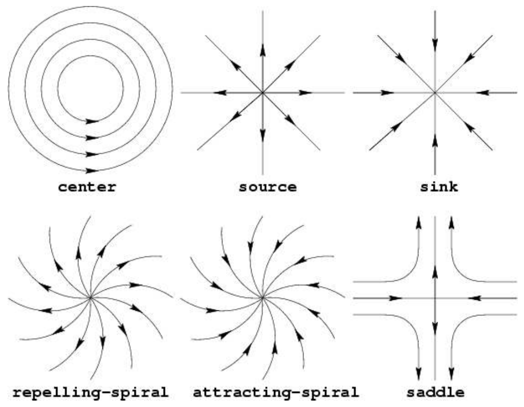

Figure 5: *Different types of critical points that occur in a flow [VKP00]. The arrow heads represent the direction of vectors.*

Seeding	Sparse/Packed
# Seeds	Small
# Steps	Large
Analysis	Store particle location for each step
Termination	1. User specified duration 2. User specified distance
Output	Connect stored location for a particle to create particle trajectories

2.3.2. Streamsurface and Pathsurface

A streamsurface is a 3D generalization of a streamline. In practice, streamsurfaces begin with a seeding curve. The particles on the seeding curve are advected for equal duration. The trajectories generated by neighboring particles are triangulated to represent a surface. From a theoretical perspective, Hultquist defines them as a locus of an infinite set of streamlines rooted at every point along a continuous line segment [Hul92]. Pathsurfaces are an extension of a streamsurface with a temporal dimension added. Streamsurfaces and pathsurfaces are very useful in studying vortices in a flow. An example of a streamsurface is presented in Figure 4.

Seeding	Seeding Curves
# Seeds	Medium
# Steps	Large
Analysis	Store particle location for each step; measure separation between neighboring particles to introduce or purge particles
Termination	1. User specified duration 2. User specified distance 3. Separation between particles is smaller than a certain threshold
Output	Triangulate/tessellate the particle locations to create a surface

2.3.3. Finite Time Lyapunov Exponents

Finite Time Lyapunov Exponents (FTLE) is a technique to calculate the rate of separation between trajectories that start close to one another. FTLE can help in identification of flow features like Lagrangian Coherent Structures (LCS) and critical points.

Lagrangian Coherent Structures and Critical Points:
Lagrangian Coherent Structures are the most repelling, attracting, and shearing surfaces in a flow that persist over time and exert a major influence on the trajectories around it [HY00]. A critical point is a feature in the flow where the magnitude of velocity is zero. Critical points are usually classified into three classes —

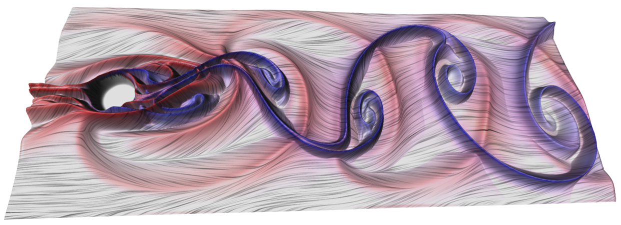

Figure 6: *Attracting (blue) and repelling (red) LCSs extracted as FTLE ridges from a two-dimensional simulation of a von Karman vortex street [KPH* 10]*

sources, sinks, and saddle points. These features can be distinguished from each other by observing the eigenvalue of the Jacobian at that point.

The FTLE algorithm produces a scalar field that is computed in two phases: particle advection and field generation.

During the particle advection phase, a set of uniformly sampled particles are placed in the volume. Each particle trajectory encapsulates the underlying flow field that transports a particle from one position to another over some finite amount of time. Collectively, all the particle trajectories form a flow map $\mathbb{R}\times\mathbb{R}\times\mathbb{R}^d \rightarrow \mathbb{R}_d : x \mapsto \phi_{t_0}^{t_0+t}(x)$, where a position $x \in \mathbb{R}^d$ at time t_0 is mapped to a new position over the time interval of duration t.

The particle trajectories are then used in the second phase, to generate the FTLE field. Using the flow map from the first phase, the Cauchy-Green tensor $\delta \in \mathbb{R}_{d\times d}$ is calculated for each particle as:

$$\delta = \frac{d\phi_{t_0}^{t_0+t}(x)}{dx} \times \frac{d\phi_{t_0}^{t_0+t}(x)}{dx}. \tag{4}$$

The FTLE value for a position x for the interval $[t_0, t_0 + t]$ is calculated using the largest eigenvalue $\lambda_{max} \in \mathbb{R}$ of the Cauchy-Green tensor weighted as:

$$\sigma_{t_0}^{t_0+t}(x) = \frac{log(\lambda_{max}(\delta))}{2\times t}. \tag{5}$$

Seeding	Packed (uniformly sampled)
# Seeds	Large
# Steps	Small
Analysis	None
Termination	1. Duration specified for FTLE calculation
Output	Calculate the exponents using the start and end locations of the particle to produce the scalar FTLE field for further analysis

2.3.4. Poincaré Map

A Poincaré plot is also referred to as a *first recurrence map* or a *return map*. It is defined as an intersection of a periodic orbit in a continuous dynamical system with a certain lower-dimensional subspace called the Poincaré section. The Poincaré section is traverse to the flow of the system. While using particle advection, the Poincaré section is usually a plane, and the Poincaré plot keeps track of the locations where particles intersect with the plane over time. The trajectories of the particle have for a Poincaré plot have to loop back around, exhibiting a periodic behavior.

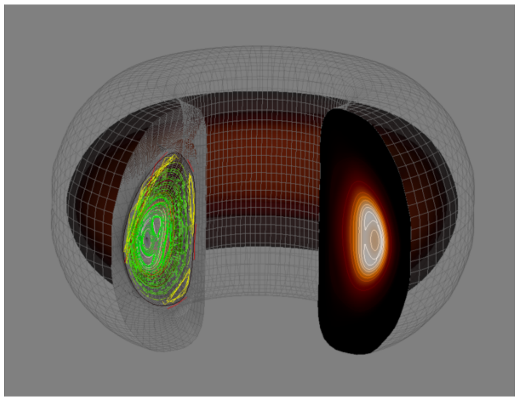

Figure 7: *The left cut away shows a Poincaré plot. The right cut away shows the iso-temperature contours. Data is from a NIMROD M3D simulation and courtesy of Scott Kruger [KSS05]*

Seeding	Packed (along a plane)
# Seeds	Medium
# Steps	Large
Analysis	Check the intersection particle steps with the planes defined for Poincaré map
Termination	1. Number of orbits for the particle
Output	The stored intersection locations for the puncture plot are used for further analysis

3. Workload Mapping

This section considers how to translate a particle advection workload into actual execution times. The section is divided into three parts. Section 3.1 discusses cost models, both in general and as applied to scientific visualization. Section 3.2 provides a formulation for assessing the types and amounts of operations performed during particle advection, using the concepts described in Section 2.2. Section 3.3 provides some nascent preliminary work on whether an analytical cost model can be accurate.

3.1. Cost Models for Scientific Visualization

Cost models are used to predict execution time of a certain algorithm. There are three different ways to define a cost model:

- Empirical cost model considers experimental results.

Solver	Data set type	$solve$	$locate$	$interp$	$terminate$	Total
Euler	Uniform	6	15	15	5	41
	Rectilinear	6	17	15	5	43
	Unstructured	6	***120***	15	5	***146***
RK4	Uniform	37	15	15	5	162
	Rectilinear	37	17	15	5	170
	Unstructured	37	***120***	15	5	***582***

Table 2: *Analytical cost calculation for particle advection. The costs in the table are by reference of a* 50^3 *grid. Hence, for the next two equations,* $d = 50$*. Rectilinear location costs :* $3 \times log(d)$*. Unstructured location costs :* $log(d^3)$*. The costs for unstructured grid are highlighted in red as the precise costs in floating point operations is difficult to predict and these are an estimate.*

- Analytical cost model considers mathematical formulation of the algorithm.
- When either of the above two are insufficient, a combination of the two can be used.

Cost models can be challenging to formulate.

There is a dearth of cost modeling work in scientific visualization. In the case of particle advection, we term this as ***Limitation 1***, since no existing work can be used to predict performance. Outside particle advection, there have been a few notable works.

Larsen et al. [LHK*16] defined a cost model for in situ rendering. The cost predicted from the model can be used manage the trade-off simulation and visualization resources during runtime. They used a combination for the empirical and analytical model for their cost predictions. The demonstrated the use of their model at large scale and to answer in situ feasibility questions.

Bruder et al. [BMFE19] presented a performance modeling of volume rendering and particle rendering. Their objective was to identify which are the most important performance factors and how these factors correlate. Their experiments, however, were only focused on GPUs and not out-of-core handling. They plan to extend their approach for performance modeling of parallel and out-of-core techniques and towards modeling not only execution time, but also energy usage and memory consumption.

3.2. Cost Formulation

This section considers the cost of particle advection from the perspective of the building blocks used to carry out particle advection. If $Cost$ denotes the particle advection costs, then a coarse formulation of $Cost$ is:

$$Cost = \sum_{i=0}^{i=P} \sum_{j=0}^{j=N_i} advance_{i,j} \tag{6}$$

where P represents the total number of particles used for the flow visualization, N_i represents the total number of steps taken by the i^{th} particle, and $advance_{i,j}$ represents the amount of work required by particle i at step j in the process of advancing the particle.

In the remainder of this subsection, we consider how this coarse formulation can be decomposed further to better illuminate the overall costs. We begin this process by exploring the process behind $advance_{i,j}$. In particular, each step that advances a particle contains three components — taking an advection step, analyzing the step in a way specific to the individual flow visualization algorithm, and checking if the particle should be terminated. Hence, Equation 6 can be written as:

$$Cost = \sum_{i=0}^{i=P} \sum_{j=0}^{j=N_i} \left(step_{i,j} + analyze_{i,j} + term_{i,j}\right) \tag{7}$$

where $step_{i,j}$ is the cost for advecting, $analyze_{i,j}$ is the cost for analyzing, and $term_{i,j}$ is the cost for the checking termination for the i^{th} particle at the j^{th} step.

The cost can be further broken down by exploring the cost for a single advection step, $step_{i,j}$. Particle advection uses an ODE solver to determine the next position of a particle. Further, the ODE solver requires the velocity of the particle at the current location. However, depending on the ODE solver, multiple velocity evaluations at locations closer to the particle may be required. The Euler solver (Equation 2) requires only one velocity evaluation, while the RK4 solver (Equation 3) requires four velocity evaluations. In all, the cost of a single particle advection step can be written as:

$$step_{i,j} = solve_{i,j} + \sum_{k=0}^{k=K} eval_{i,j,k} \tag{8}$$

where $solve_{i,j}$ is the cost for the ODE solver to determine the next position, K is the number of velocity evaluations required by the ODE, and $eval_{i,j,k}$ is the cost for velocity evaluation for the i^{th} particle for the $j^{th} step$ at the k^{th} location.

The cost for velocity evaluations, $eval_{i,j,k}$, can be further broken down into two more components. Each evaluation involves two operations: locating the current cell for the current evaluation, and interpolating the velocity values for the current position using velocities at the vertices of the current cell. In all, the cost of velocity evaluations can be written as:

$$eval_{i,j,k} = locate_{i,j,k} + interp_{i,j,k} \tag{9}$$

where $locate_{i,j,k}$ is the cost for locating the cell, and $interp_{i,j,k}$ is the cost for interpolating the velocities at the k^{th} location.

Further, we can substitute 9 in 8 to yield:

$$step_{i,j} = solve_{i,j} + \sum_{k=0}^{k=K} (locate_{i,j,k} + interp_{i,j,k}) \tag{10}$$

Finally, we can substitute 10 in 7 to obtain our final formulation:

$$Cost = \sum_{i=0}^{i=P} \sum_{j=0}^{j=N_i} \Big(solve_{i,j} + \sum_{k=0}^{k=K} (locate_{i,j,k} + interp_{i,j,k}) + analyze_{i,j} + term_{i,j} \Big) \tag{11}$$

3.3. Preliminary Work: Analytical Cost Modeling

This section presents preliminary work of cost modeling for particle advection in two parts. Section 3.3.1 demonstrates cost estimation for a hypothetical workload based on Equation 11. Section 3.3.2 presents experimental validation for costs estimated using the method in Section 3.3.1.

3.3.1. Cost Estimation

To understand the process of cost estimation, consider a hypothetical usage of a flow visualization algorithm advancing a million particles in a 3D uniform grid for a maximum of 1000 steps. Further assume this usage employs a fourth order Runge-Kutta solver. Then, the cost of each component can be estimated as follows:

Locate: The locate operation in a uniform grid uses 15 FLOP to find the cell and the particle's location within the cell for interpolation.
Interpolate: The interpolate operation in a uniform grid requires trilinear interpolation which uses 15 FLOP to evaluate the velocity at the given location.
Solve: The solve operation for the RK4 integration scheme uses 37 FLOP to calculate the determine the next position of the particle.
Analyze: The cost of analysis of the step varies based on the visualization technique being used. In case only deals with advancing particles and hence the analysis cost is 0.
Terminate: The terminate operation requires 5 FLOP to determine if the particle is outside the spatio-temporal bounds or if the particles completes the maximum number of steps.

These costs can be substituted in Equation 11 to get the final cost of a single step of a particle in the presented situation.

$$\begin{aligned} Cost &= \sum_{i=0}^{i=1M} \sum_{j=0}^{j=1000} \Big(37 + \sum_{k=0}^{k=4} (15+15) + 0 + 5 \Big) \\ &= \sum_{i=0}^{i=1M} \sum_{j=0}^{j=1000} \Big(37 + 120 + 0 + 5 \Big) \\ &= \sum_{i=0}^{i=1M} \sum_{j=0}^{j=1000} 162 \\ &= \sum_{i=0}^{i=1M} 162,000 \\ &= 162,000,000,000 \text{ FLOP} \end{aligned} \tag{12}$$

Table 2 presents the cost of each of the individual components for the RK4 solver and the Euler solver across the three commonly used types of meshes. This table can be used to calculate the cost of a particle advection workload in terms of floating point operations (FLOP) for most flow visualization use cases.

3.3.2. Empirical Validation

The cost formulation in Section 3.2 and the method to estimate the cost for a workload in Section 3.3.1 are only useful if they can successfully predict the execution time for a given workload. This section presents a set of experiments performed where the calculated cost is translated into an execution time for a workload and them compared against the its actual execution time. These experiments were performed on an Intel Xeon E5-1650 CPU with a clock rate of 3.80 GHz using a particle advection implementation from the VTK-m visualization library [MSU*16] with a single CPU core. The data used for the experiments was of the resolution $50 \times 50 \times 50$ enabling comparison with the estimated FLOP costs in Table 2.

The results of the experiments are presented in Table 3. The cost factor defined in the last column is the ratio of the total executed FLOP from the experiment (product of execution time and clock speed of the CPU) and the estimated FLOP using Equation 11. The cost predicted using the equation can be determined valid if this factor stays constant for all experiments. In the case of this study this factor stays constant between 12 and 13 across all experiments using uniform and rectilinear data and the Euler and RK4 solvers. However, even if this factor is consistent for all experiments using unstructured grids, it differs from the other experiments. This is because the cost function is not comprehensive and fails to account for the contributions of accessing and indexing large arrays typical of cell locators for unstructured grids. In all, these experiments show that an analytical approach may be quite difficult, and that at least some elements of an empirical model may be needed. As a result, this nascent preliminary work is unable to address Limitation 1 and additional research will be needed to address this topic.

Data type	Solver	Number of Particles	Number of Steps	Cost Eq. 11	Cost VTK-m	Cost Factor
Uniform	Euler	1000	1000	41e6	0.131	12.13
		10000	1000	410e6	1.311	12.15
		100000	1000	4100e6	13.161	12.20
	RK4	1000	1000	162e6	0.519	12.18
		10000	1000	1620e6	5.197	12.19
		100000	1000	16200e6	51.910	12.18
Rectilinear	Euler	1000	1000	74e6	0.147	12.98
		10000	1000	740e6	1.455	12.86
		100000	1000	7400e6	14.745	13.03
	RK4	1000	1000	294e6	0.574	12.82
		10000	1000	2940e6	5.808	12.98
		100000	1000	24900e6	57.298	13.03
Unstructured	Euler	1000	1000	146e6	4.657	121.21
		10000	1000	1460e6	49.682	129.31
	RK4	1000	1000	582e6	18.653	121.79
		10000	1000	5820e6	188.029	122.77

Table 3: *Comparing analytical cost and actual execution cost for particle advection. The first four columns describe the workload. The fifth column, labeled “Cost Eq. 11” is the estimated number of FLOPS for this workload using information from Table 2 and our cost formula (Equation 11). The sixth column, labeled “Cost VTK-m” is the execution time of running this workload on a single core using the VTK-m software. The final column calculates the cost factor, i.e., the factor relating the number of FLOPS with the actual execution time. The experiments were run a single CPU core running at 3.8GHz, and issuing at most one floating-point operation per cycle, meaning a potential of 3.8 GFLOPS. Taking the first row as an example, this workload ran for 0.131s meaning it had the potential to issue 497 MFLOPS (i.e.,* $0.131s \times 3.8$ *GFLOPS). The cost equation indicated that this workload only needed to issue 41 MFLOPS, so the resulting cost factor is 12.13 (i.e., 497 MFLOPS / 41 MFLOPS). A consistent cost factor across experiments would build confidence that an analytic approach can be used to estimate runtimes. That said, while this nascent attempt shows consistency across rectilinear and uniform meshes, unstructured meshes have very different values.*

4. Algorithmic Optimizations

This section surveys algorithmic optimizations for particle advection building blocks, i.e., techniques for executing a given building using fewer operations. Some of the building blocks do not particularly lend themselves to algorithmic optimizations. For example, a Runge-Kutta solver requires a fixed number of FLOPS, and the only possible “optimization” would be to use a different solver. That said, cell location allows room for possible optimizations. Further, the efficiency of vector field evaluation can be improved by considering underlying I/O operations. These two optimizations are discussed in Sections 4.1 (cell location) and 4.2 (I/O efficiency).

4.1. Cell Locators

Cell locators operate by first storing all of the cells in a data structure and then repeatedly locating which cell or cells contain a point. The storage step is done as a pre-processing step, typically before advection begins. This storage process often requires dedicated memory and can be time consuming. In a typical scenario, for N cells, the storage is $O(N)$ bytes and the execution time is $O(NlogN)$. That said, these pre-processing costs can be offset by improved performance in location, which typically takes no additional storage and can be performed in $O(logN)$ time. In practice, cell locators quickly improve over the naïve approach, which would iterate through the cells one at a time to find the containing cell and thus take $O(N)$ time. – even for a few particles traveling a short distance, a cell locator offers performance benefits.

There is a storage-performance trade-off when using cell locators for large meshes, i.e., if the data structure is able to store data at a very fine level, the cell location can be made to execute faster. In practice, it is not always possible to have enough memory for large data structures. In particular, storage can be a critical problem for GPUs with limited memory. In such cases, one solution is for a data structure to store cells at some coarse level in a list, i.e., the last level of the data structure may return a list of “candidate cells” that may contain the location L_i instead of a single cell. An additional performance penalty for searching through the list of candidate cells would then be incurred to identify the containing cell for L_i.

According to Lohner and Ambrosiano [LA90] the process of cell location can follow one of the following three approaches.

Using a Cartesian background grid: This approach superimposes an unstructured mesh onto a regular grid (Cartesian grid).

A linked list is created for each cell in the regular grid, and each linked list contains the unstructured cells that lie within the corresponding regular grid's cell. The cell location problem is reduced to 1) calculating which cell contains the point in the Cartesian grid, and 2) traversing the associated linked list to find the actual containing cell for the point. However, this approach suffers when the large variances in cell sizes for the background mesh may introduce inefficiencies and inaccuracies. If the number of cells employed for the background grid is too small, the linked list traversal can become restrictive. If the number of cells employed for the background grid is too large, the storage overhead can become restrictive.

Using tree structures: This approach solves the previous approach's shortcomings by using a hierarchy of Cartesian meshes, for example by using octrees. The hierarchical nature better accommodates large meshes that have significant variations in cell shapes and sizes. However, Lohner and Ambrosiano note that vectorization of this approach is challenging as tree-based schemes introduce additional indirect addressing.

Using successive neighbor searches: This approach carries out a search by using information from previous searches. This approach hinges on the assumption that the particle does not travel too far away from its previous position. If the required location for velocity evaluation is not present in the previously identified host cell, the immediate neighbors for the cell are likely to contain it. The assumption is justified by the accuracy and stability requirements of particle advection, and thus the number of cells needed to be searched is small. However, the first search for a particle has no previous information to draw from. Hence, the first search must use a different technique. For successive searches, the neighboring cells are searched until the host cell is identified. However, Lohner and Ambrosiano note that vectorization is difficult due to variance in the number of cells considered for a given particle.

Lohner and Ambrosiano also presented an algorithm and ways to vectorize the cell location process using a successive neighbor search method. Initially, the search is performed using all particles, to vectorize the process better. As the process continues, particles that still do not have a containing cell are grouped together at the top of the list, and the search is only repeated over these particles. They also study the algorithm in multiprocessor environments where the particle list is subdivided, and each processor is assigned a subgroup of particles. Their vectorized version of the algorithm demonstrated a speedup of 14× over an unvectorized version.

Ueng et al. [USM96] also adopted the successive neighbor search method to cell location in particle advection for efficient streamline, streamribbon, and streamtube construction. They restricted their work to linear tetrahedral cells for simplification of certain formulations. However, this requires the algorithm to apply a preprocessing step for the decomposition of an unstructured mesh into tetrahedra. When applied to tetrahedral meshes, the successive neighbor search approach is sometimes also referred to as ***tetrahedral walk*** [BRKE*11].

Kenwright and Lane [KL96] expand the work by Ueng et al. by improving the technique to identify the particle's containing tetrahedron. Their approach uses fewer floating point operations for cell location compared to Ueng et al. The cell location approach maps the particle's physical coordinates to Barycentric coordinates within the containing tetrahedron. Their experiments were performed in a curvilinear grid, with the grid's hexahedral cells decomposed into tetrahedral cells on the fly. For the initial cell location operation and in cases where a particle travels further than the immediate neighboring cells, the ***boundary search*** technique described by Buning [Bun89] was used. In other cases, the Barycentric coordinates and a lookup table were used for efficient cell location. Kenwright and Lane reported performance improvement of 6× compared to strategies that work over the physical space.

Sadarjeon et al. compared particle advection in ***C-space*** or ***computational space*** against ***P-space*** or ***physical space*** [SVWHP94]. P-space algorithms work directly over irregular grids. C-space algorithms rely on a Cartesian grid, different from the approach described by Lohner and Ambrosiano. Instead of using the Cartesian grid in the background to simplify cell location, the irregular grids are transformed into a Cartesian grid to simplify numerical procedures. However, this approach requires transform operations between the two spaces which introduce inefficiencies — 1) transforming the corner velocities of hexahedral element from P-space to C-space, and 2) transforming the output position of every advection step from C-space to P-space. The rest of the algorithm (including interpolation and integration) proceeds in the C-space. Sadarjeon et al. note that C-space algorithms are computationally more expensive than P-space algorithms. However, their work does not provide a performance comparison of the two approaches.

Schiriski et al. [SBK06] presented a GPU-based approach for particle advection that uses tetrahedral walk. Their approach used a kd-tree to perform the initial search for a containing cell of a particle. However, searching the kd-tree is done using the CPU.

Bußler et al. [BRKE*11], and Garth and Joy [GJ10] also presented approaches for particle advection where cell location relies solely on GPUs. These are discussed in more detail in Section 5.1

4.2. I/O Efficiency

Simulations with very large numbers of cells often output their vector fields a block-decomposed fashion, such that each block is small enough to fit in the memory of a compute node. Flow visualization algorithms that process block-decomposed data vary in strategy, although many operate by storing a few of these blocks in memory at a time, and loading/purging blocks as necessary. This method of computation is known as out-of-core computation. One of the significant bottlenecks for flow visualization algorithms while performing out-of-core computations is the cost of I/O. Particle advection is a data-dependent operation and efficient prefetching to ensure sequential access to data can be very beneficial in minimizing these I/O costs This section discusses the works that aim to improve particle advection performance by improving the the efficiency of I/O operations.

Algorithm	Application	Intent / Evaluation	Data Size	Time Steps	Seed Count	Performance
Lohner and Ambrosiano [LA90]	Streamlines	Fast cell location and efficient vectorization	870^*	-	10K	14×
Ueng et al. [USM96]	Streamlines	Streamline computation and cell location in canonical coordinate space	$320K^*$	-	100	1.61×
			$225K^*$	-		1.59×
			$288K^*$	-		1.58×
Chen et al. [CXLS11]	Streamlines	Improving data layout for better I/O performance	134M	-	4K	0.96 – 1.30×
			200M	-		0.98 – 1.98×
			537M	-		0.99 – 1.29×
Chen et al. [CNLS12]	Pathlines	Improving data layout for better I/O performance	25M	48	4K	1.25 – 1.38×
			65M	29		1.10 – 1.31×
			80M	25		1.19 – 1.36×
Chen et al. [CS13]	FTLE	Improving data layout for better I/O performance	25M	48	-	1.08 – 1.32×
			65M	29		
			80M	25		

Table 4: *Summary of studies considering algorithmic improvements to particle advection. The asterisk for entries in the data size column represent unstructured grids.*

Chen et al. [CXLS11] presented an approach to improve the I/O efficiency of particle advection for out-of-core computation. Their approach relies on constructing an access dependency graph (ADG) based on the flow data. The graphâĂŹs nodes represent the data blocks, and the edges are weighted based on the probability that a particle travels from one block to another. While advecting particles, runtime decisions for prefetching data blocks to minimize data block misses are made using the data layout obtained using a cost model, which used the information from the graph. The authors demonstrated significant speed-ups based on their method to layout data against the H-curve data layout. In the worse case, the ADG-based layout performed at par with the H-curve layout, and in the best case, the ADG-based layout outperformed the H-curve layout by 98%.

Chen et al. [CNLS12] extended the previous work to out-of-core computation of pathlines. For comparison, they used a Z-order space-filling (Z-curve) layout. Their results show a performance improvement in the range of 10%-40% compared to the Z-curve layout.

Chen et al. [CS13] expanded the work further to introduce a seed scheduling strategy to be used along with the graph-based data layout. They demonstrated an efficient out-of-core approach to calculate FTLE. The performance improvements observed against Z-curve layout were in the range of 8%-32%.

4.3. Summary

Table 4 presents the summary of studies that address algorithmic optimizations. Optimizations to cell locators for unstructured grid enable significant speed-ups for the workloads. With a combination of efficient cell location and vectorization Lohner and Ambrosiano achieved the speed-up of 14×. However, the other study demonstrated a speed-up of around 1.6×. The works by Chen et al. for efficient I/O for particle advection all demonstrated speed-ups up to 1.3×

Solver	Data type	Particles	Steps	Scaling CPU	Scaling GPU
Euler	Uniform	1000	1000	5.48	3.29
		10000	1000	7.15	9.65
		100000	1000	7.38	11.02
	Rectilinear	1000	1000	4.84	2.76
		10000	1000	6.19	8.04
		100000	1000	6.44	9.23
	Unstructured	1000	1000	9.02	1.28
		10000	1000	9.53	4.19
RK4	Uniform	1000	1000	6.89	3.90
		10000	1000	7.49	10.46
		100000	1000	7.57	11.54
	Rectilinear	1000	1000	6.09	3.06
		10000	1000	6.69	8.49
		100000	1000	6.68	9.23
	Unstructured	1000	1000	9.06	1.29
		10000	1000	9.02	3.95

Table 5: *Shared memory parallelism cost summarization.*

5. Optimizations for Shared Memory Setting

This section discusses flow visualization works that address the performance challenges in a shared memory environment. Shared memory parallelism refers to using parallel resources on a single node. The devices that enable shared memory parallelism are multi- and many-core CPUs and other accelerators, such as GPUs. In the case of shared memory parallelism, multiple threads of a program running on different cores of a processor (CPU or a GPU) share memory, hence the nomenclature. One of the primary reasons for the increase in supercomputers' compute power can be attributed to the advancements of CPUs and accelerator hardware. In all, for applications to make cost-effective use of resources, it has become exceedingly important to use shared memory resources efficiently. However, making efficient use creates many challenges for the programmers and users. Two important factors to consider are 1) efficient use of concurrency, and 2) performance portability.

5.1. Shared Memory Particle Advection Using GPUs

GPUs have become a popular accelerator choice in the past decade, with most leading supercomputers using GPUs as accelerators [top20]. Part of this has been the availability of specialized toolkits, including early efforts like Brook-GPU [BFH*04] and popular efforts like Nvidia's CUDA [Nic07], that enable GPUs to be used as general purpose computing devices [BBC*08]. However, programming applications for efficient execution on a GPU remains challenging for three main reasons. First, unlike CPUs which are built for low latency, GPUs are built for high throughput. CPUs have fewer than a hundred cores, while GPUs have a few thousand. However, each CPU core is significantly more powerful than a single GPU core. Second, efficient use of the GPU requires applications to have sufficiently large parallel workloads. Third, executing a workload on a GPU also has an implicit cost of data movement between the host and the device, where a host is the CPU and the DRAM of the system, and the device is the GPU and its dedicated memory. This cost makes GPUs inefficient for smaller workloads.

This section discusses particle advection using GPUs in two parts. Section 5.1.1 discusses works that use GPUs for interactive flow visualization, using particle advection results for rendering directly from the GPU. Section 5.1.2 discusses works that use CPUs for improving the performance of cell location for particle advection.

5.1.1. Particle Advection on GPU

Particle advection can benefit from using GPUs when there are many particles to advect. As particles can be advected independently of one another, each particle can be scheduled with a separate thread of the GPU, making the most of the available concurrency. Many works have tried to address performance issues of particle advection using GPUs, however, with different goals.

Krüger et al. [KKKW05] presented an approach for interactive visualization of particles in a steady flow field using a GPU. They exploited the GPU's ability to simultaneously perform advection and render results without moving the data between the CPU and the CPU. This was done by accessing the texture maps in the GPU's vertex units and writing the advection result. Their approach on the GPU provided the interactive rendering at 41 fps (frames per second) compared to 0.5 fps on the CPU.

Bürger et al. [BSK*07] extended the particle advection framework described by Krüger et al. for unsteady flow fields. With their method, unsteady data is streamed to the GPU using a ring-buffer. While the particles are being advected in some time interval $[t_i, t_{i+1}]$, another host thread is responsible for moving t_{i+2} from host memory to device memory. At any time, up to three timesteps of data are stored on the device. By decoupling the visualization and data management tasks, particle advection and visualization can occur without delays due to data loading. Bürger et al. [BKKW08] further demonstrated the efficacy of their particle tracing framework for visualizing an array of flow features. These features were gathered using some metric of importance, e.g., FTLE, vorticity, helicity, etc.

Bürger et al. [BFTW09] also provided a way for interactively rendering streak surfaces. Using GPUs, the streak surfaces can be adaptively refined/coarsened while still maintaining interactivity.

The works summarized above aim to make efficient use of GPU for interactive flow visualization. They use GPUs to directly interact with the rendering artifacts, saving efforts to move data between the host and the devices.

5.1.2. Cell Location on GPUs

Bußler et al. [BRKE*11] presented a GPU-based tetrahedral walk for particle advection. Their approach for cell location borrows heavily from the work by Schiriski et al. [SBK06] discussed in Section 4.1. However, they can execute the cell location strategy entirely on the GPU and do not require the CPU for the initial search. Additionally, they evaluated different Kd-tree traversal strategies to evaluate the impact of these strategies on the tetrahedral walk Their results conclude that the ***single-pass*** method, which performs only one pass through the kd-tree to find the nearest cell vertex (without the guarantee of it being the nearest) performs the best. The other strategies evaluated in the study were ***random restart*** and ***backtracking***.

Garth and Joy [GJ10] presented an approach for cell location based on bounding interval hierarchies. Their search structure, called ***celltree***, improves construction times via a heuristic to determine good spatial partitions. The authors presented a use case of advecting a million particles on a GPU in an unstructured grid with roughly 23 million hexahedral elements. The celltree data structure was able to obtain good performance on GPUs despite no GPU-specific optimizations.

5.2. Other Shared Memory Particle Advection Works

Most published works on shared-memory parallelism with CPUs is in the context of hybrid parallelism, i.e., using a combination of shared- and distributed-memory parallel techniques. For these works, the distributed-memory elements managed dividing work among nodes, and the shared-memory parallelism ap-

Algorithm	Application	Intent / Evaluation	Data Size	Time Steps	Seed Count	Performance
Krüger et al. [KKKW05]	source-dest	Interactive flow visualization (steady) using GPUs	-	-	-	60-80×
Bürger et al. [BSK*07]	various	Interactive flow visualization (unsteady)				
Bürger et al. [BKKW08]	various	Interactive flow visualization using importance metrics	7M 4M 1M	- 22 30		
Bürger et al. [BFTW09]	streak surface	Interactive streak surface visualization	589K 4.1M	102 22	400	
Schirski et al. [SBK06]	pathlines, source-dest	Efficient cell location on GPUs	$0.8M^*$ $1.1M^*$ $3.7M^*$	5 101 200	1M	
Garth et al. [GJ10]	source-dest	Efficient cell location on GPUs for unstructured grids / Comparison against CPUs	$23.6M^*$	-	250K 1M	16.5×
Bußler et al. [BRKE*11]	source-dest	Efficient cell location on GPUs using improved tetrahedral walk	$4.2M^*$ $115M^*$ $743M^*$	5 101 200	1M	
Pugmire et al. [PYK*18]	source-dest	Performance Portability / Comparison with specialized comparators for CPUs and GPUs	134M 134M 134M	-	10M	$0.37-0.48\times$ (GPUs) $0.29-0.36\times$ (CPUs) $1.56-2.24\times$ (GPUs) $0.79-0.84\times$ (CPUs) $1.42-2.04\times$ (GPUs) $0.51-0.59\times$ (CPUs)

Table 6: *Summary of shared memory particle advection. The asterisk for entries in the data size column represent unstructured grids.*

proach was providing a "pool" of cores that could advect particles quickly. Specifically:

- Camp et al. [CGC*11b] presented two approaches that used multi-core processors to parallelize particle advection 1) parallelization over particles, and 2) parallelize over data blocks. In both cases, the authors aimed to use the N allocated cores. For parallelization over particles, N worker threads were used along with N I/O threads. The worker threads are responsible for performing particle advection. The I/O threads manage the cache of data blocks to support the worker threads. For parallelization over data blocks, $N-1$ worker threads are used, which access the cache of data blocks directly, and an additional thread was used for communicating results with other processes.
- Camp et al. [CKP*13] also extended their previous work to GPUs. One of their objectives was to compare particle advection performance on the GPU against CPU under different workloads. They varied the datasets, the number of particles, and the duration of advection for their experiments. Their findings suggest that in the case where the workloads have fewer particles or longer durations, the CPU performed better. However, in most cases otherwise, the GPU was able to outperform the CPU.
- Childs et al. [CBP*14] explored particle advection performance across various GPUs (counts and device) and CPUs (processors and concurrency). Their objective was to explore the relationship between parallel device choice and the execution time for particle advection. Two of their key findings were: 1) For CPUs, adding more cores benefited workloads that execute for medium to longer duration, 2) CPUs with many cores were as performant as GPUs and often outperformed GPUs when the execution times for the tests was short.

Pugmire et al. [PYK*18] provided a platform portable solution for particle advection using the VTK-m library. The solution builds on data parallel primitives provided by VTK-m. Their results demonstrated very good platform portability, providing comparable performance to platform specific solutions on many-core CPUs and Nvidia GPUs.

5.3. Synthesis

In terms of published research, Table 6 presents a summary of shared memory particle advection. These studies either presented approaches for interactive flow visualization or optimizations for particle advection of GPUs using cell locators, with one exception that demonstrated platform portability.

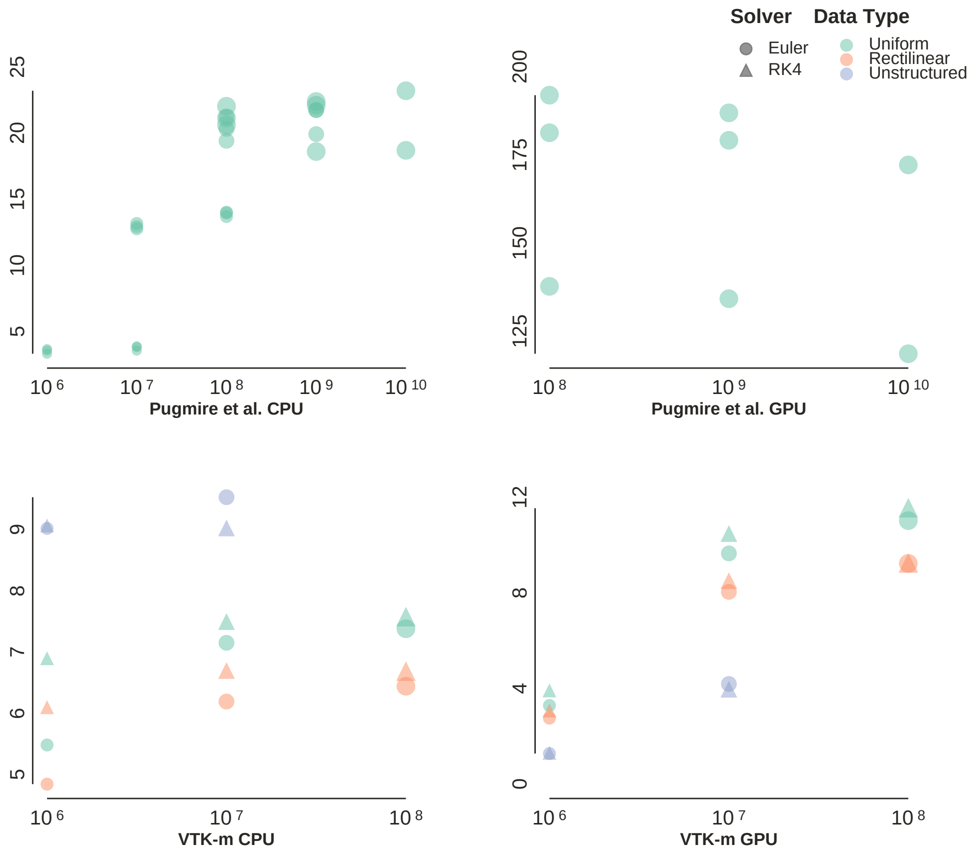

Figure 8: *Scatter plots for CPU and GPU speed-ups for particle advections workloads. The X axis represents the magnitude of the workload in terms of total number of steps for each of the sub plots and the Y axis represents the speed-up. The size of the glyphs corresponds to the number of particles used in the experiment. The data for the plots on the top is collected from numbers reported by Pugmire et al. [PYK*18] They used 28 CPU cores and a Nvidia P100 GPU. The data from the plots on the bottom is collected from new experiments using VTK-m. This study used 12 CPU cores and a Nvidia K80 GPU.*

Figure 8 shows a preliminary result for understanding the characteristics of shared memory parallel particle advection. Some of the key observations are:

1. Workloads with more number of particles scale better with added parallelism for the same amount of total work.
2. The studies that used the RK4 integrator generally scaled better than the ones that used the Euler integrator.
3. The experiments with unstructured data scaled better on the CPU than on the GPU. This could be because of the nature of memory accesses required by cell locators and justifies more research into GPU based cell locators.

Additionally, the plots for the CPUs demonstrate consistency in terms of scalability when the workload is increased. The plot for the P100 GPUs (top right) suggests that it is not able to scale larger workloads with the same efficiency as the smaller workloads considered by Pugmire et al. [PYK*18] There is also a tremendous variation in the speed-ups achieved by two considered GPUs, where the P100 GPU is able to achieve speed-ups of over 125× and the K80 GPU achieves speed-ups of less than 12×. This points to ***Limitation 2*** of existing literature. The performance difference of particle advection between two generations of GPUs can be significant. Existing studies fail to capture this relation and makes it harder to estimate to speed-up that can be realized. Understanding the performance characteristics of particle advection across different GPUs is planned as future work.

6. Optimizations for Distributed Memory Setting

This section summarizes large-scale parallel particle advection-based flow visualization studies. Computational fluid dynamics simulations are capable of producing large volumes of data. Analyzing such volumes of data to extract useful information demands resources equivalent to that of the simulation. In most cases, this means access to many nodes of a supercomputer to handle the computational and memory needs of the analysis. Solutions that employ particle advection often execute in a distributed memory setting. The objective of the distribution of work is to perform efficient computation, memory and I/O operation, and communication. There are multiple strategies for distributing particle advection workloads in a distributed memory setting to achieve these objectives. These can be categorized under two main classes:

Parallelize over particles: Particles are distributed among parallel processes. Each process only advances particles assigned to it. Data is typically loaded as required for each of the particles.

Parallelize over data: Blocks of partitioned data are distributed among parallel processes. Each process only advances particles that occur within the data blocks assigned to it. Particles are communicated between processes based on their data requirement.

Most distributed particle advection solutions are either an optimization of over these two classes or a combination of these two classes. The decision to choose between these two classes depends on multiple factors, of which Camp et al. [CGC*11b] identify the most prominent to be:

The volume of data set: Data output by simulation can span the spectrum from small to large. While small data sets can fit in the memory of a single node, large data sets have to be partitioned and distributed among for them to be processed. If the data set can fit in memory, it can be easily replicated across nodes and particles can be distributed among nodes, i.e., the work can be parallelized over particles. However, for large partitioned data sets, work parallelized over data is can be more efficient.

The number of particles: Different particle advection based visualization techniques have different work requirements. Some require small number of particles integrated over a long duration, while others require a large number of particles advanced for a short duration. In the case where fewer particles are needed, parallelization over data is a better approach as it could potentially reduce I/O costs. In the case where more particles are needed, parallelization over particles can help better distribute computational costs.

Distribution of particles: The placement of particles for advection can potentially cause performance problems. When using parallelization over data, if particles are concentrated within a small region of the data set, the processes owning the associated data blocks will be responsible for a lot of computation while most other processes remain idle. Parallelization over particles can lead to better work distribution in such cases.

Data set complexity: The characteristics of the vector field have a significant influence on the work for the processes, e.g., if a process owning a data block that contains a sink, most particles will advect towards it, causing the process to do more work than the others. In such a case, parallelize over particles will enable better load balance. On the other hand, when particles switch data blocks often (e.g., a circular vector field), parallelize over data is better since it reduces the costs of I/O to load required blocks.

This section describes distributed particle advection works in two parts. Section 6.1 describes the optimization for parallelizing distributed particle advection in more depth. Section 6.2 summarizes finding from the survey of distributed particle advection studies.

6.1. Parallelization Methods

This section presents distributed particle advection works in three parts. Section 6.1.1 presents works that optimize parallelization over data. Section 6.1.2 presents works that optimize parallelization over particles. Section 6.1.3 presents works that use a combination of parallelization over data and particles.

6.1.1. Parallelization over data

"Parallelize over data" is a paradigm for work distribution in flow visualization where M data blocks are distributed among N processors. Each process is responsible for performing computations for active particles within the data blocks assigned to them. This method aims to reduce the cost of I/O operations, which is more expensive than the cost of performing computations.

Sujudi and Haimes [SH96] elicit the problems introduced by decomposing data into smaller blocks that can be used within the working memory of a single node. They present the seminal work in generating streamlines in a distributed memory setting using the parallelize over particles scheme. They used a typical client-server model where clients perform the work, and the server coordinates the work. Clients are responsible for the computation of streamlines within their sub-domain; if a particle hits the boundary of the sub-domain, it requests the server to transfer the streamline to the process that owns the next sub-domain. The server is responsible for keeping track of client request and sending streamlines across to the clients with the correct sub-domain. No details of the method used to decompose the data in sub-domains are provided.

Camp et al. [CGC*11b] compared the MPI-only implementation to the MPI-hybrid implementation of parallelizing over data. They noticed that the MPI-hybrid version benefits from reduced communication of streamlines across processes and increased throughput when using multiple cores to advance streamlines within data blocks. Their results show between 1.5x-6x improvement in the overall times for the MPI-hybrid version over the MPI-only version. The parallelize over data scheme is sensitive to the distribution of particles and complexity of vector field. The presence of critical points in certain blocks of data can potentially lead to load imbalances. Several techniques have been developed to deal with such cases and can be classified into two categories 1) works that require knowledge of vector field discussed in Section 6.1.1.1, and 2) works that do not require knowledge of vector field discussed in Section 6.1.1.2.

6.1.1.1. Knowledge of vector field required The works classified in this category acquire knowledge of vector fields by performing a pre-processing step. Pre-processing allows for either data or particles to be distributed such that all processes perform the same amount of computation.

Chen et al. presented a method that employs repartitioning of the data based on flow direction, flow features, and the number of particles [CF08]. They perform pre-processing of the vector field using various statistical and topological methods to enable effective partitioning. The objective of their work is to produce partitions such that the streamlines produced would seldom have to travel between different data blocks. This enabled them to speed up the computation of streamlines due to the reduced communication between processes.

Yu et al. [YWM07] presented another method that relies on pre-processing the vector field. They treat their spatiotemporal data as 4D data instead of considering the space and time dimensions as separate. They perform adaptive refinement of the 4D data using a higher resolution for regions with flow features and a lower resolution for others. Later, cells in this adaptive grid are cluster hierarchically using a binary cluster tree based on the similarity of cells in a neighborhood. This hierarchical clustering helps them to partition data that ensure workload balance. It also enables them to render pathlines at different levels of abstraction.

Nouanesengsy et al. [NLS11] used pre-processing to estimate the workload for each data block by advecting the initial set of particles. The estimates calculated from this step are used to distribute the work among processes. Their proposed solution maintained load balance and improved performance. While the solutions in this category are better at load balancing, they introduce an additional step of pre-processing which has its costs. This cost may be expensive and undesirable if the volume of data is significant.

6.1.1.2. Knowledge of vector field not required The works classified in this category aim to balance load dynamically without any pre-processing.

Peterka et al. [PRN*11] performed a study to analyze the effects of data partitioning on the performance of particle tracing. Their study compared static round-robin (also known as block-cyclic) partitioning to dynamic geometric repartitioning. The study concluded that while static round-robin assignment provided good load balancing for random dense distribution of particles, it fails to provide load balancing when data blocks contain critical points. They also noticed that dynamic repartitioning based on workload could improve the execution time between 5% to 25%. However, the costs to perform the repartitioning are restrictive. They suggest more research needs to focus on using less synchronous communication and improvements in computational load balancing.

Nouanesengsy et al. [NLL*12] extended the work by Perterka et al. to develop a solution for calculating Finite-Time Lyapunov Exponents (FTLE) for large time-varying data. The major cost in performing FTLE calculations is incurred due to particle tracing. Along with *parallelize over data*, they also used *parallelize over time*, which enabled them to create a pipeline that could advect particles in multiple time intervals in parallel. Although their work did not focus on load-balancing among processes, it presented a novel way to optimize time-varying particle tracing. Their work solidifies the conclusions about static data partitioning of the study by Peterka et al.

Zhang et al. [ZGH*17] proposed a method that does very well in achieving dynamic load balancing. Their approach used a new method for domain decomposition, which they term as the constrained K-d tree. Initially, they decompose the data using the K-d tree approach such that there is no overlap in the partitioned data. The partitioned data is then expanded to include ghost regions to the extent that it still fits in memory. Later, the overlapping areas between data blocks become regions to place the splitting plane to repartition data such that each block gets an equal number of particles. Their results show better load balance was achieved among processes without additional costs of pre-processing and expensive communication. Their results also demonstrate higher parallel efficiency. However, their work makes two crucial assumptions 1) an equal number of particles in data blocks might translate to equal work, and 2) the constrained K-d tree decomposition leads to an even distribution of particles. These assumptions do not always hold practically.

In conclusion, pre-processing works can achieve load balance with an additional cost for *parallelize over data*. This cost goes up with large volumes of data. The overall time for completing particle advection might not benefit from the additional cost of pre-processing, especially when the workload is not compute-intensive. Most solutions that rely on dynamic load balancing suffer from increased communication costs or are affected by the distribution of particles and the complexity of the vector field. The work proposed by Zhang et al. is promising but still does not guarantee optimal load balancing.

6.1.2. Parallelize over particles

"Parallelize over particles" is a paradigm for work distribution in flow visualization where M particles are distributed among N processors. Most commonly, the particle distribution is done such that each process is responsible for computing the trajectories of $\frac{M}{N}$ particles. Each process is responsible for the computation of streamlines for particles assigned to it. This is done by loading the data blocks required by the process in order to advect the particles. Particles are advected until they can no longer continue within the current data block, in which case another data block is requested and loaded.

Previous works have explored different approaches to optimize the scheme described above. Since the blocks of data are loaded whenever requested, the cost of I/O is a dominant factor in the total time. Prefetching of data involves predicting the next needed data block while continuing to advect particles in the current block to hide the I/O cost. Most commonly, predictions are made by observing the I/O access patterns. Rhodes et al. [RTBS05] used these access patterns as a priori knowledge for caching and prefetching to improve I/O performance dynamically. Akande et al. [AR13] extended their work to unstructured grids. The performance of these methods depends on making correct predictions of the required blocks. One way to improve the prediction accuracy is by using a graph-based approach to model the dependencies between data blocks. Some works use a preprocessing step to construct these graphs [CXLS11, CNLS12, CS13]. Guo et al. [GZL*14] used the access dependencies to produce fine-grained partitions that could be loaded at runtime for better efficiency of data accesses.

Zhang et al. [ZGY16] presented an idea of higher-order access transitions, which produce a more accurate prediction of data accesses. They incorporated historical data access information to calculate access dependencies.

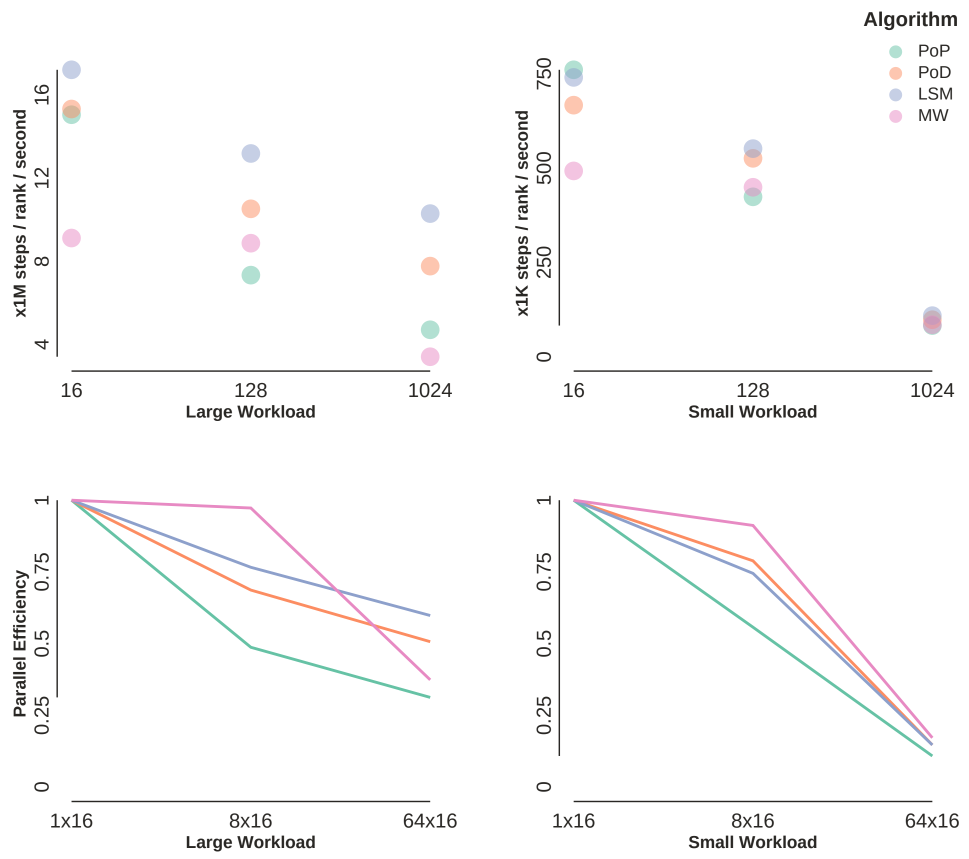

Figure 9: *Weak scaling plots for distributed memory particle advection based on the comparison of four parallelization algorithms by Binyahib et al. [BPYC20]. The plots present performance comparison of the algorithms for two different workloads. The large workload used 1 particle per 100 cells where each particle advanced 10K steps. The small workload used 1 particle per 10K cells where each particle advanced 1K steps. The plots on the top present the performance of the algorithms in terms of throughput along the Y axis, while the plots on the bottom present the parallel efficiency while using weak scaling along the Y axis. In all cases the X axis represents the number of MPI ranks used to perform the experiments.*

Since particles assigned to a single process might require access to different blocks of data, most of the works using parallelization over particles use a cache to hold multiple data blocks. The process advects all the particles that occur within the blocks of data currently present in the cache. When it is no longer possible to continue computation with the data in the cache, blocks of data are purged, and new blocks are loaded into the cache. Different purging schemes are employed by these methods, among which “Least-Recently Used,” or LRU is most common. Lu et al. [LSP14] demonstrated the benefits of using a cache in their work for generating stream surfaces. They also performed a cache-performance trade-off study to determine the optimal size of the cache.

Camp et al. [CGC*11b] presented work comparing the MPI only and MPI-hybrid implementations of parallelizing over particles. Their objective was to prove the efficacy of using shared memory parallelism with distributed memory to reduce communication and I/O costs. They observed 2x-10x improvement in the overall time for calculation of streamlines while using the MPI-hybrid version.

Along with caching, Camp et al. [CCC*11] also presented work that leverages different memory hierarchies available on modern supercomputers to improve the performance of particle advection. The objective of the work is to reduce to cost of I/O operations. Their work used Solid State Drives (SSDs) and local disks to store data blocks, where SSDs are used as a cache. Since the cache can only hold limited amounts of data compared to local disks, blocks are purged using the LRU method. When required blocks are not in the cache, the required data is searched in local disks before accessing the file system. The extended hierarchy allows for a larger than usual cache, reducing the need to perform expensive I/O operations.

One trait that makes the parallel computation of integral curves challenging is the dynamic data dependency. The data required to compute the curve cannot be determined in advance. However, this information is crucial for optimal load-balanced parallel scheduling. One solution to this problem is to opt for dynamic scheduling. Two well-studied techniques for dynamic scheduling are *work-stealing* and *work-requesting*. In both approaches, an idle process acquires work from a busy process. Popularly, idle processes are referred to as thieves, and busy processes are referred to as victims. The major distinction between work-stealing and work requesting is how the thief from the victim acquires the work. In work-requesting, the thief requests work items, and the victim voluntarily shares it. In work-stealing, the thief directly accesses the victimâĂŹs queue for work items without the victim knowing.

A huge body of works addresses *work-stealing* in *task-based* parallel systems in general [BL99, DLS*09, ST19]. In the case of integral curve calculation, task-based parallelism inspires the parallelize over particles scheme. In work-stealing, idle processes select busy processes at random and steal work from them. Dinan et al. [DLS*09] demonstrated the scalability of the work-stealing approach. Work stealing can be readily applied to the parallelize over particles approach. Lu et al. [LSP14] presented a technique for calculating stream surface efficiently using work-stealing. Instead of using particles as a work item, they used segments of the seeding curve for the stream surface. Work stealing has been proven to be efficient in theory and practice. However, Dinan et al. report its implementation is complicated.

Muller et al. [MCHG13] presented an approach that uses work requesting for tracing particle trajectories. Their algorithms start by equally distributing all work items (particles) among processes. However, they started by assigning all particles to a single process for performing the load balancing study. Every time an active particle from the work queue is unable to continue in the currently cached data, it is placed at the end of the queue. Whenever a thief tries to request work, the particles from the end of the queue are provided, reducing the current processesâĂŹ need to load the data block for the particle. The results reported performance improvements between 30% to 60%.

According to Childs et al. [CPA*10], the dominant factor affecting the performance of parallelizing over particles is I/O. The solution to solve the I/O problem during runtime is to perform prefetching of data. However, works that propose prefetching incur additional costs of making predictions of which blocks to read. Leveraging the memory hierarchy similar to Camp et al. is a good strategy, provided proper considerations for vector field size and complexity are made. Apart from I/O costs, load balancing remains another factor affecting performance adversely. Previous work stealing and work requesting strategies have demonstrated good load balance with additional costs of communicating work items. These costs could potentially be restrictive in the case of workloads with a large number of particles.

6.1.3. Hybrid Parallelism

The works described in this section combine *parallelize over data* and *parallelize over particles* schemes to achieve optimal load balance. Pugmire et al. [PCG*09] introduced an algorithm that uses a master-slave model. The processes were divided into groups, and each group had a master process. The master is responsible for maintaining the load balance between processes as it coordinates the assignment of work. The algorithm begins with statically partitioning the data. All processes load data on demand. Whenever a process needs to load data for advancing its particles, it coordinates with the master. The master decides whether it is more efficient for the process to load data or to send its particles to another process. The method proved to be more efficient in I/O and communication than the traditional parallelization approaches.

Problem Classification	Parallelization Strategy	
	over data	over particles
Dataset size	Large	Small
Number of particles	Small	Large
Seed Distribution	Sparse	Dense
Vector Field Complexity	No critical points	No circular field

Table 7: *Recommendation of parallelization strategy for particle advection workloads based on features of the problem. This table appears in the survey by Binyahib [Bin19].*

Kendall et al. [KWA*11] provided a hybrid solution which they call DStep and works like the MapReduce framework. Their algorithm uses groups for processes as well and has a master to coordinate work among different groups. A static round-robin partitioning strategy is used to assign data blocks to processes, similar to Peterka et al. [PRN*11]. The work of tracking particles is split among groups where the master process maintains a work queue and assigns work to processes in its group. Processors within a group can communicate particles among them. However, particles across groups can only be communicated by the master processes. The algorithm provided an efficient and scalable solution for particle tracing and has been used by other works [GYHZ13, GHS*14, LGZY16].

Lu et al. [LSP14] introduced another hybrid approach. Their algorithm aims for the efficient generation of stream surfaces. The seeding curve for streamlines was divided into segments, and these segments were assigned to processes as tasks. In their implementation, each process maintains a queue of segments. When advancing the streamline segment using the front advancing algorithm proposed by Garth et al. [GTS*04], if a segment starts to diverge, it is split into two and placed back in the queue. The work-stealing approach lets idle processes acquire work from the queues of busy processes. When a processor requires additional data to advance a segment, it requests the data from the processes that own the data block. The work demonstrated good load balancing and scalability.

6.2. Synthesis

This section summarizes distributed particle advection in two parts. First, general take-aways are discussed based on the various factors discussed in the introduction of this section. Second, observations from the studies in terms of their particle advection workloads are presented.

Algorithm	Architecture	Procs.	Data set Size	Time steps	Seed Count	Application	Seeding Strategy	Intent / Evaluation
Yu et al. [YWM07]	Intel Xeon (8x4)	32	644M	-	1M	streamlines, pathlines	-	hierarchical representation, strong scaling
	AMD Optron (2048x2)	256	644M	100	1M		-	
Chen et al. [CF08]	Intel Xeon (48x2)	32	162M	-	700	streamlines	-	data partitioning / strong scaling
Pugmire et al. [PCG*09]	Cray XT5 (ORNL) (149K)	512	512M	-	4k, 22K	streamlines	uniform	data loading, data partitioning / weak scaling
		512	512M	-	10K		uniform	
		512	512M	-	20K		uniform	
Peterka et al. [PRN*11]	PowerPC-450 (40960x4)	16k	8B	-	128k	streamlines,	random	domain decomposition, dynamic repartitioning / strong and weak scaling
		32K	1.2B	32	16k	pathlines	random	
Camp et al. [CCC*11]	Intel Xeon Dash (SDSC)	-	512M	-	2.5K, 10K	streamlines	dense, uniform	Effects of storage hierarchy
		-	512M	-	2.5K, 10K			
		-	512M	-	2.5K, 10K			
Camp et al. [CGC*11b]	Cray XT4 (NERSC) 9572x4	128	512M	-	2.5K, 10K	streamlines	dense, uniform	MPI-hybrid parallelism
		128	512M	-	2.5K, 10K			
		128	512M	-	1.5K, 6K			
Nouanesengsy et al. [NLS11]	PowerPC-450 (1024x4)	4K	2B	-	256K	streamlines	random	workload aware domain decomposition / strong and weak scaling
		4K	1.2B	-	128K		random	
Nouanesengsy et al. [NLL*12]	PowerPC-450 (40960x4)	1k	8M	29	186M	FTLE	uniform	pipelined temporal advection, caching / strong and weak scaling
		1K	25M	48	65.2M		uniform	
		16k	345M	36	288M		uniform	
		16K	43.5M	50	62M		uniform	
Camp et al. [CCG*12]	Cray XT4 (NERSC) (9572x4)	128	512M	-	128	stream surface	rake	Comparison of parallelization algorithms for stream surfaces
		128	512M	-	361		rake	
		128	512M	-	128		rake	
Muller et al. [MCHG13]	AMD Magny-Cours (6384x24)	1K	32M	735	1M	streamlines, pathlines	uniform	work requesting / load balancing, strong scaling
Childs et al. [CBP*14]	Nvidia Kepler (1 GPU / Proc)	8	1B	-	8M	source-dest	uniform	Distriburted particle advection over different hardware architectures / comparison, strong scaling
	Intel Xeon	192	1B	-	8M			
Guo et al. [GZL*14]	Intel Xeon (8x8)	64	755M	100	-	streak surface,	seed line	sparse data management / strong scaling
		64	3.75M	24	200	pathlines,	uniform	
	Intel Xeon (700x12)	512	25M	48	-	FTLE	uniform	
Lu et al. [LSP14]	PowerPC A2 (2048x16)	1K	25M	-	32K	stream surface	rakes	caching, performance / strong scaling
		4K	80M	-	32K		rakes	
		8K	500M	-	32K		rakes	
		8K	2B	-	64K		rakes	
Zhang et al. [ZGY16]	Intel Xeon (8x8)	64	3.75M	24	6250	pathlines	uniform	data prefetching / strong scaling
		64	25M	48	4096		-	
Zhang et al. [ZGH*17]	PowerPC A2 (2048x16)	8K	1B	-	128M	streamlines,	-	domain decomposition, using K-d trees / strong and weak scaling
		8K	3.8M	24	8M	source-dest,	-	
		8K	25M	48	24M	FTLE	uniform	

Algorithm	Architecture	Procs.	Data set Size	Time steps	Seed Count	Application	Seeding Strategy	Intent / Evaluation
Binyahib et al. [BPC19]	Intel Xeon (2388x32)	512	67M	-	1M	source-dest	dense, uniform	In situ parallelization over particles

Table 8: *Summary of large scale distributed particle advection worklets. The numbers in parenthesis in the Architecture column represent the total number of cores available on the execution platform.*

Application	Particles / 1k Cells
Souce-destination	72222.20
FTLE	5013.02
Streamlines	9.89
Pathlines	6.93
Stream surface	0.25

Table 9: *Number of particles used per one thousand cells of data for different applications from works described in Table 8.*

Table 7 provides a simple lookup for a parallelization strategy based on various workload factors discussed earlier in the section. These strategies were presented in a survey by Binyahib [Bin19]. ***Parallelize over data*** is best suited when the data set volume of large. In the presence of flow features like critical points and vortices, it suffers from load imbalances. While several methods have been proposed for data partitioning for load-balanced computation, these works incur the cost of preprocessing and redistributing data. ***Parallelize over particles*** is best suited when the number of particles is large. It suffers from load imbalances due to inconsistencies in the computational work for different particles. Some works aim to address the problem of load imbalances but have added costs of pre-processing, communication, and I/O. ***Hybrid*** solutions demonstrate better scalability and efficiency compared to the traditional methods. However, implementing these methods is very complicated and typically has some added cost of communication and I/O.

Figure 9 presents a comparison of scaling behaviors of four parallelization algorithms, extracted from the study presented by Binyahib et al. [BPYC20]. These algorithms include parallelize over particles, parallelize over data, Lifeline Scheduling Method (LSM, an extension of parallelize over particles), and master worker (a hybrid parallel algorithm). This study presents a weak scaling of these algorithms. The top row plots show the throughput of these algorithms in terms of number of steps completed by each MPI rank per second. The bottom row plots show the efficiency of weak scaling achieved by the different algorithms. The efficiency of the algorithms drop significantly as the concurrency and workload is increased. The drop is more significant in smaller workloads than in larger workloads. This points to ***Limitation 3*** of the existing literature. The only study which compares the scaling behaviors of the most widely used parallelization algorithm uses weak scaling. In order to be able to quantify the speed-ups resulting from added distributed parallelism for a given workload, a strong scaling study is necessary. The strong scaling study for these algorithms is a potential avenue for future research.

Table 8 summarizes large-scale parallel particle advection-based flow visualization studies in terms of the distributed executions and the magnitudes of the workloads. The platforms used by the considered studies in this section span from desktop computers to large supercomputers. The work with the least amount of processes and work in this survey is by Chen et al. [CF08], which used only 32 processes to produced 700 streamlines. The work with the most number of processes was by Nouanesengsy et al. [NLL*12], which used 16 thousand processes for FTLE calculation. However, the work with the most work was by Binyahib et al. [BPYC20], which used 34 billion particles for advection.

Table 9 summarizes the number of particles used in proportion to the size of the data used in the works included in Table 8. Stream surface generation is the application that required the least amount of particles. A significant part of the cost of generating stream surfaces comes from triangulating the surfaces from the advected streamlines. These streamlines cannot be numerous as they may lead to issues like occlusion. Source-destination queries use the most particles in proportion to the data size. All other applications need to store a lot of information in addition to the final location of the particle — streamlines and pathlines need to save intermediate locations for representing the trajectories, stream surfaces need the triangulated surface for rendering, and FTLE analysis needs to generate an additional scalar field. Source-destination analysis has no such costs and can instead use the savings in storage and computation to incorporate more particles.

7. Performance Portability

It is very common for users to run experiments on multiple computers. Quiet commonly simulations are run on one cluster and analysis is run on other. Also, when new generation of architectures replace existing ones, applications need to be ported to execute with greater or at least the same efficiency on newer hardware. It is not desirable for users to rewrite platform-specific, optimized versions of code when such cases occur. Hence, platform portability has become an important issue in saving costs incured due to porting applications on newer hardware devices. These problems are being addresed by programming libraries like Kokkos [ET13], RAJA [HK14], and VTK-m [MSU*16].

The remainder of this section draws from a study I participated in on the VTK-m software and its approach for portable performance. The study has received a minor revision from the journal Parallel Computing, and should be published later this year. I performed all of the survey work described in this section, including extracting all data for the tables and figures, and making the tables and figures. That said, much of the text was by my co-authors on the paper.

7.1. VTK-m

To evaluate VTK-m, we surveyed ten studies on VTK-m collate their results. In all cases, these studies considered a single visualization algorithm in depth. We feel this provides a holistic picture of VTK-m as it evaluates a considerable number of algorithms, platforms, data sets, and comparators. We divide our results into two sections. The first section (7.1.1) evaluates how the hardware-agnostic approach of VTK-m versus hardware-specific comparators for nine different algorithms. The second section (7.1.2) considers studies that focused on scaling properties on CPU hardware — when given more cores, did performance improve proportionally? This section considers five such studies, although four of the studies also had hardware-specific comparators and appeared in section 7.1.1.

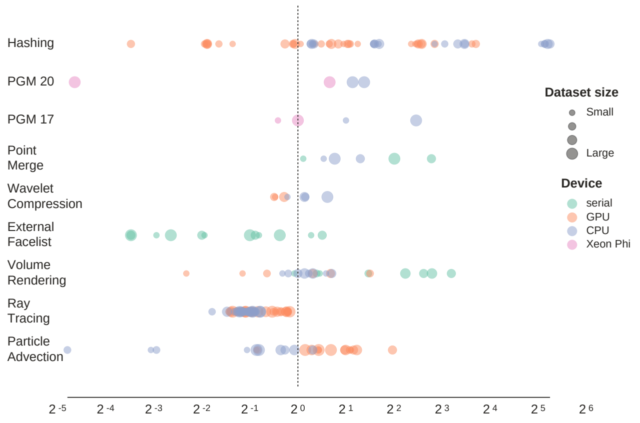

Figure 10: *VTK-m against hardware-specific comparators for nine algorithms. The horizontal axis represents the ratio in VTK-m against its comparator — 2^{-1} means VTK-m twice as long, 2^1 means VTK-m half the time, and 2^0 (1) means VTK-m the same amount of time. The circle glyphs indicate both the hardware type (circle color) and the size of the data set being processed (circle size). For data size, all plotting is relative — the largest data set considered in that study gets the largest circle size, the smallest data set gets the smallest circle size, and the other circle glyphs are scaled proportionally between the two extremes.*

7.1.1. VTK-m Hardware-Specific Comparators

Table 10 and Figure 10 show the results from nine studies comparing individual algorithms. The hardware-specific comparators came from a mix of well-known visualization/rendering software (Embree [W*14], HAVS [CICS05], OptiX [P*10], VisIt [C*12], VTK [SML96], Vapor [CMNR07b]) and direct implementations (CUDA, OpenMP, pthreads, TBB, Thrust). While Table 10 lists the range of comparisons, Figure 10 shows the outcome for each individual experiment.

Two of the algorithms, external facelist and ray tracing, had significantly worse performance with VTK-m than with its hardware-specific comparators. For external facelist, the cause was an asymmetric comparison — the comparator was a serial algorithm that could skip any overhead necessary for any parallel algorithm. Specifically, the serial comparator could update its internal tables without fear of collisions, whereas the VTK-m had to be designed to prevent collisions when running in parallel. For ray tracing, the cause was that the comparators are extremely efficient — NVIDIA's OptiX and Intel's Embree are the product of dedicated teams embedded at their respective hardware vendors. The difference in performance, then, is likely more a reflection of respective development time than a statement about VTK-m Further, the VTK-m actually beat OptiX on older NVIDIA cards, speaking to the extent that OptiX is tuned for the latest NVIDIA hardware and to the portable performance of VTK-m

Table 11 summarizes aggregate performance over the nine algorithms from their respective studies, and we use the methodology for calculating individual table entries as our best estimate at relative performance. Four of the nine algorithms are faster than their hardware comparators, whereas five are slower. This provides evidence that VTK-m good performance, as the expected outcome if VTK-m as good as hardware-specific implementation would be a 50/50 mix of faster and slower. On the hardware side, we saw that serial and CPU experiments were faster. This could possibly demonstrate a benefit of DPP programming, as it is not possible to incorporate serial bottlenecks. We felt the GPU performance of 0.95X was quite good, and matches our team's experiences with good GPU performance. Similarly, we have felt that Xeon Phi performance was poor, which is borne out in the table. In particular, the study by Perciano et al. [PHC*20] demonstrated poor performance at high scale. VTK-m was competitive at lower concurrency, but not when the hyperthreads were involved. This may indicate a shortcoming in Xeon Phi device adapters. Finally, the aggregation of the experiments demonstrate a 1.14X speedup from using VTK-m hardware-specific comparators. As previously discussed, these gains are coming from serial and multi-core CPU improvements, and likely from limiting programmers to only using fast programming constructs. That said, the GPU results (0.95X) indicate that VTK-m competitive.

7.1.2. VTK-m on Multi-Core CPUs

For each of the studies that ran scaling studies on multi-core CPUs (not Xeon Phi), Table 12 shows the parallel efficiency and Figure 11 shows the behavior with increasing numbers of cores. For the most part, scaling is quite good, as the VTK-m get proportionally faster as more cores are added. The two exceptions are for contour trees (where scalability is similar for a native OpenMP implementation due to algorithm complexities) and for the point merge algorithm when it switches to hyperthreading. That said, we find the overall scalability is good evidence that VTK-m an effective programming paradigm for multi-core CPUs.

Algorithm	Architecture	Comparator	Performance
Threshold [MMA*13]	Intel Xeon X5680 (1C)	VTK	3.6
	Intel Xeon X5680 (12C)	custom	4
	Nvidia Tesla C2050	custom	26
Ray Tracing [L*15]	Intel I7 4770K	Intel Embree	0.28 – 0.48
	Intel Xeon E5-2680 v2	Intel Embree	0.4 – 0.58
	Nvidia GTX Titan Black (Kepler)	Nvidia OptiX	0.44 – 0.56
	Nvidia Tesla K80M (Kepler)	Nvidia OptiX	0.37 – 0.51
	Nvidia GeForce GTX 750Ti	Nvidia OptiZ	0.69 – 0.89
	Nvidia GeForce GT 620M	Nvidia OptiX	0.73 – 1.16
Volume Rendering [LLN*15]	Intel Ivy Bridge/NERSC Edison (1C)	VisIt	0.73 – 9.1
	Intel Ivy Bridge/NERSC Edison (24C)	self-1C	17.5
	Intel I7 4770K (8C, 4 physical)	VTK [BKS97]	2
	Nvidia GTX Titan Black (Kepler)	HAVS	0.33 – 2
Halo and center finding [SLH*15]	Intel Xeon E5-2670 (16C)	custom	no comparison
	Nvidia Tesla M2090	custom-16C MPI	2.5 – 4.9
	Intel Xeon Phi SE10P (MIC)	custom-60C MPI	0.13
	AMD Opteron (16C)	custom	no comparison
	Nvidia Tesla K20x	custom-32C MPI	6
External facelist calculation [LBMC16]	Intel Xeon E5-2650 v2 (1C)	VTK, VisIt	0.50 – 1.4, 0.08 – 0.25
	Intel Xeon E5-2650 v2 (16C)	self-1C	8.8 – 12
	Nvidia Tesla K40 (Kepler)	self-16C	0.9 – 2.25
Contour tree computation [CWSA16]	Intel Xeon E5- 4650L (1C)	custom	0.6
	Intel Xeon E5- 4650L (32C)	custom-1C, self-1C	13.1, 9.2
	Nvidia Tesla K40m (Kepler)	custom-1C, self-1C	21.0, 14.8
Wavelet Compression [L*17]	Intel Haswell (16C)	Vapor [CR05, CMNR07a]	0.8 – 1.5×
	Nvidia Tesla K40 (Kepler)	custom CUDA	0.6 – 0.8
Maximal clique enumeration [LPM*17]	Intel Xeon(R) E5-2667v3 (16C)	custom-1C	0.05 – 7.4
	NVIDIA Tesla K40 (Keplar)	custom-1C	0.05 – 11.6
Particle Advection [P*18]	Intel Xeon E5-2650 (16C)	VisIt	2.4 – 3.6
	Intel Xeon E5-2695 v3 (28C)	pthreads [CGC*11a]	0.03 ∗ −1.6
	IBM Power8 (20C)	pthreads	0.05 ∗ −1.03
	Nvidia Tesla K20x (Kepler)	CUDA [CKP*13]	0.37 – 2.26
	Nvidia Tesla K80 (Kepler)	CUDA	0.54 – 2.45
	Nvidia Tesla P100 (Pascal)	CUDA	0.48 – 4.08
Point Merge [YCM19]	IBM Power9 (1C)	VTK	1.07 – 6.86
	IBM Power9 (40C)	VTK	1.4 – 2.5
	Nvidia Tesla V100 (Volta)	VTK-m	0.48 – 4.0
Probablistic Graphical Modeling [L*18]	Intel Ivy Bridge (24C)	custom	2 – 7
	Intel Xeon Phi 7250 (68C)	custom	0.75 – 4.25
	Nvidia Tesla K40 (Kepler)	self-KNL	3.5 – 4.14
Probablistic Graphical Modeling [PHC*20]	Intel Xeon E5-2609 v2 (8C)	custom (OpenMP, threads)	2.2, 2.6
	Intel Xeon Phi 7250 (68C)		1.25, 5.93

Table 10: *Studies comparing performance for algorithms implemented in VTK-m against hardware-specific implementations. In the architecture column, numbers in parentheses specify number of multi-core CPU cores used in the experiments. Finally, in the performance column, the numbers represent the range of outcomes. For example, VTK-m ray tracing algorithm only had 28% to 48% of the performance (i.e., from almost 4X slower to nearly 2X slower) compared to Intel's Embree ray tracer when run on the Intel I7 architecture.*

Algorithm	CPUs	GPUs	X. Phi	Serial	Total
External facelist	-	-	-	0.34	0.34
PGM 18	3.32	-	0.87	-	1.69
PGM 20	2.39	-	0.25	-	0.78
Particle advection	0.38	1.53	-	-	0.76
Point merge	1.82	-	-	3.10	2.38
Ray tracing	0.47	0.55	-	-	0.51
Volume rendering	1.13	0.83	-	3.10	1.43
Wavelet compression	1.13	0.75	-	-	0.92
Hashing	5.97	1.45	-	-	2.94
Total	1.45	0.95	0.47	1.48	**1.14**

Table 11: *Aggregate performance of VTK-m against hardware-specific comparators for nine algorithms on four hardware architectures. Each entry in the table represents a geometric mean over its experiments. For example, if algorithm X on hardware Y had three experiments, where VTK-m five times as fast, half as fast, and one fourth as fast as its hardware comparator, then the table will contain 0.85 (= $(5 \times 0.5 \times 0.25)^{\frac{1}{3}}$). We use the geometric mean since it captures aggregate behavior better than an arithmetic mean (which would be 1.92 for the previous example). We also combined results in algorithm and in hardware, i.e., if algorithms X, Y, and Z had results on hardware W, we calculated performance on W as the geometric mean of results for X, Y, and Z on W. One benefit of this approach is that is unaffected by the number of experiments run for a given study — if one study contained one hundred experiments and another study contained two, then the findings from the first study do not overshadow the second. Finally, we calculated the geometric mean over all hardware-algorithm pairs, which was 1.14.*

Algorithm	Architecture	Max Cores	Parallel Efficiency
Volume Rendering [LLN*15]	I Ivy Bridge	24	0.73
External Facelist [LBMC16]	I Ivy Bridge	16	0.77
Contour Tree [CWSA16]	I Sandy Bridge	32	0.24
Point Merge [YCM19]	IB Power 9	40	0.55
Particle Advection [P*18]	I Haswell	28	0.78

Table 12: *Surveying multi-core CPU scaling studies for five visualization algorithms. While these studies considered many concurrency levels, this table shows the maximum concurrency, as well as the parallel efficiency achieved at that maximum concurrency.*

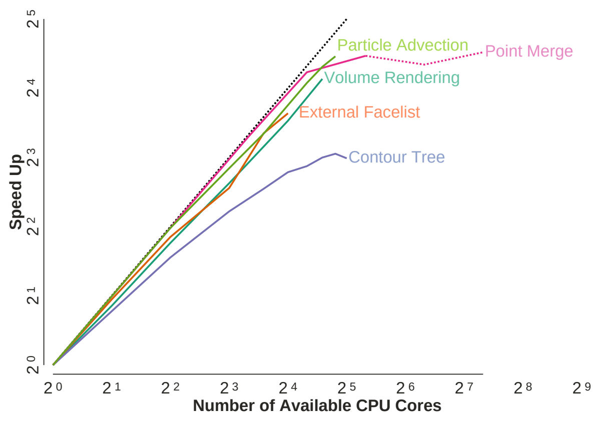

Figure 11: *Plotting scaling study results for five visualization algorithms on multi-core CPUs. Each visualization algorithm is plotted as its own line, with a unique color. When an algorithm uses hyperthreads, the line becomes dotted. Finally, the ideal scaling line is drawn as a dotted black line.*

8. Conclusion

This survey has endeavored to inform a workflow for deciding whether to employ optimizations for particle advection-based flow visualization. It considered aspects of this workflow involving work estimation and work optimization. In particular, Sections 2 and 3 provided a formalization of how different aspects of particle advection interact and a formula counting the number of operations, which are potentially useful for future work in formal cost modeling. It also synthesized relevant works on optimizations, whether algorithmic or via parallelism, and contributed some new understanding on gaps in our community's understanding with respect to speedups via parallelism.

A distinct contribution is in the organization of the components of a flow visualization system, as seen in Figure 2. These components suggest an opportunity to develop an efficient software design for a grand unified particle advection framework.

While this survey has informed a decision-making workflow for particle advection workloads, current limitations prevent it from being realized at this time. These limitations are:

1. None of the existing research presents cost estimation for various operations related to particle advection. While some are easy to predict, predicting the costs for unstructured grids is challenging and more research is needed.
2. For shared memory, the amount of speed-ups for workloads on CPUs was consistent. But the observed speed-ups on GPUs varied significantly. It is challenging to predict how much a

GPU will speed-up a certain workload. A study comparing speed-ups for particle advection across multiple GPUs would help make such predictions.

3. For distributed memory, the best holistic study did weak scaling. The premise of this paper is more about strong scaling. But weak scaling results are poor, so strong scaling is likely to be poorer still. Understand of strong scaling properties of distributed memory particle advection will enable better prediction of speed-ups while introducing more parallelism.

All these limitations also open avenues for future research.

References

[AR13] Akande O. O., Rhodes P. J.: Iteration aware prefetching for unstructured grids. In *2013 IEEE International Conference on Big Data* (2013), IEEE, pp. 219–227. 17

[BBC*08] Bergman K., Borkar S., Campbell D., Carlson W., Dally W., Denneau M., Franzon P., Harrod W., Hill K., Hiller J., et al.: Exascale computing study: Technology challenges in achieving exascale systems. *Defense Advanced Research Projects Agency Information Processing Techniques Office (DARPA IPTO), Tech. Rep 15* (2008). 13

[BFH*04] Buck I., Foley T., Horn D., Sugerman J., Fatahalian K., Houston M., Hanrahan P.: Brook for gpus: stream computing on graphics hardware. *ACM transactions on graphics (TOG) 23*, 3 (2004), 777–786. 13

[BFTW09] Buerger K., Ferstl F., Theisel H., Westermann R.: Interactive streak surface visualization on the gpu. *IEEE Transactions on Visualization and Computer Graphics 15*, 6 (2009), 1259–1266. 13, 14

[Bin19] Binyahib R.: Scientific visualization on supercomputers: A survey. Available at `http://www.cs.uoregon.edu/Reports/AREA-201903-Binyahib.pdf` (2019/12/12), 2019. Area Exam. 19, 21

[BKKW08] Burger K., Kondratieva P., Kruger J., Westermann R.: Importance-driven particle techniques for flow visualization. In *2008 IEEE Pacific Visualization Symposium* (2008), IEEE, pp. 71–78. 13, 14

[BKS97] Bunyk P., Kaufman A., Silva C. T.: Simple, fast, and robust ray casting of irregular grids. In *Scientific Visualization Conference (dagstuhl '97)* (1997), pp. 30–30. 23

[BL99] Blumofe R. D., Leiserson C. E.: Scheduling multithreaded computations by work stealing. *Journal of the ACM (JACM) 46*, 5 (1999), 720–748. 19

[BMFE19] Bruder V., Müller C., Frey S., Ertl T.: On evaluating runtime performance of interactive visualizations. *IEEE transactions on visualization and computer graphics 26*, 9 (2019), 2848–2862. 8

[BPC19] Binyahib R., Pugmire D., Childs H.: In situ particle advection via parallelizing over particles. In *Proceedings of the Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization* (2019), pp. 29–33. 21

[BPYC20] Binyahib R., Pugmire D., Yenpure A., Childs H.: Parallel particle advection bake-off for scientific visualization workloads. In *2020 IEEE International Conference on Cluster Computing (CLUSTER)* (2020), pp. 381–391. 18, 21

[BRKE*11] Bussler M., Rick T., Kelle-Emden A., Hentschel B., Kuhlen T. W.: Interactive particle tracing in time-varying tetrahedral grids. In *EGPGV* (2011), pp. 71–80. 11, 13, 14

[BSK*07] Bürger K., Schneider J., Kondratieva P., Krüger J. H., Westermann R.: Interactive visual exploration of unsteady 3d flows. In *EuroVis* (2007), pp. 251–258. 13, 14

[Bun89] Buning P. G.: Numerical algorithms in cfd post-processing. *von Karman Institute for Fluid Dynamics* (1989). 11

[C*12] Childs H., et al.: VisIt: An End-User Tool For Visualizing and Analyzing Very Large Data. In *High Performance Visualization–Enabling Extreme-Scale Scientific Insight*. Oct 2012, pp. 357–372. 22

[CBP*14] Childs H., Biersdorff S., Poliakoff D., Camp D., Malony A. D.: Particle advection performance over varied architectures and workloads. In *2014 21st International Conference on High Performance Computing (HiPC)* (2014), IEEE, pp. 1–10. 14, 20

[CCC*11] Camp D., Childs H., Chourasia A., Garth C., Joy K. I.: Evaluating the benefits of an extended memory hierarchy for parallel streamline algorithms. In *2011 IEEE Symposium on Large Data Analysis and Visualization* (2011), IEEE, pp. 57–64. 18, 20

[CCG*12] Camp D., Childs H., Garth C., Pugmire D., Joy K. I.: Parallel stream surface computation for large data sets. In *Ieee symposium on large data analysis and visualization (ldav)* (2012), IEEE, pp. 39–47. 20

[CF08] Chen L., Fujishiro I.: Optimizing parallel performance of streamline visualization for large distributed flow datasets. In *2008 IEEE Pacific Visualization Symposium* (2008), IEEE, pp. 87–94. 16, 20, 21

[CGC*11a] Camp D., Garth C., Childs H., Pugmire D., Joy K.: Streamline integration using mpi-hybrid parallelism on a large multicore architecture. *IEEE Transactions on Visualization and Computer Graphics 17*, 11 (2011), 1702–1713. 23

[CGC*11b] Camp D., Garth C., Childs H., Pugmire D., Joy K. I.: Streamline Integration Using MPI-Hybrid Parallelism on a Large Multicore Architecture. *IEEE Transactions on Visualization and Computer Graphics (TVCG) 17* (Nov. 2011), 1702–1713. 14, 16, 18, 20

[CICS05] Callahan S. P., Ikits M., Comba J. L. D., Silva C. T.: Hardware-Assisted Visibility Sorting for Unstructured Volume Rendering. *IEEE Transactions on Visualization and Computer Graphics 11*, 3 (2005), 285–295. 22

[CKP*13] Camp D., Krishnan H., Pugmire D., Garth C., Johnson I., Bethel E. W., Joy K. I., Childs H.: GPU Acceleration of Particle Advection Workloads in a Parallel, Distributed Memory Setting. In *Eurographics Symposium on Parallel Graphics and Visualization* (2013), The Eurographics Association. `doi:10.2312/EGPGV/EGPGV13/001-008`. 14, 23

[CMNR07a] Clyne J., Mininni P., Norton A., Rast M.: Interactive desktop analysis of high resolution simulations: application to turbulent plume dynamics and current sheet formation. *New Journal of Physics 9*, 8 (2007), 301. 23

[CMNR07b] Clyne J., Mininni P., Norton A., Rast M.: Interactive desktop analysis of high resolution simulations: application to turbulent plume dynamics and current sheet formation. *New Journal of Physics 9*, 8 (2007), 301. 22

[CNLS12] Chen C.-M., Nouanesengsy B., Lee T.-Y., Shen H.-W.: Flow-guided file layout for out-of-core pathline computation. In *IEEE Symposium on Large Data Analysis and Visualization (LDAV)* (2012), IEEE, pp. 109–112. 12, 17

[CPA*10] Childs H., Pugmire D., Ahern S., Whitlock B., Howison M., Weber G. H., Bethel E. W., et al.: Extreme scaling of production visualization software on diverse architectures. *IEEE Computer Graphics and Applications*, 3 (2010), 22–31. 19

[CR05] Clyne J., Rast M.: A prototype discovery environment for analyzing and visualizing terascale turbulent fluid flow simulations. In *Visualization and Data Analysis 2005* (2005), vol. 5669, International Society for Optics and Photonics, pp. 284–294. 23

[CS13] Chen C.-M., Shen H.-W.: Graph-based seed scheduling for out-of-core ftle and pathline computation. In *2013 IEEE symposium on large-scale data analysis and visualization (LDAV)* (2013), IEEE, pp. 15–23. 12, 17

[CWSA16] Carr H. A., Weber G. H., Sewell C. M., Ahrens J. P.: Parallel peak pruning for scalable smp contour tree computation. In *IEEE Symposium on Large Data Analysis and Visualization (LDAV)* (2016), pp. 75–84. 23, 24

[CXLS11] CHEN C.-M., XU L., LEE T.-Y., SHEN H.-W.: A flow-guided file layout for out-of-core streamline computation. In *2011 IEEE Symposium on Large Data Analysis and Visualization* (2011), IEEE, pp. 115–116. 12, 17

[DLS*09] DINAN J., LARKINS D. B., SADAYAPPAN P., KRISHNAMOORTHY S., NIEPLOCHA J.: Scalable work stealing. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis* (2009), IEEE, pp. 1–11. 19

[DP80] DORMAND J. R., PRINCE P. J.: A family of embedded runge-kutta formulae. *Journal of computational and applied mathematics 6*, 1 (1980), 19–26. 4

[ET13] EDWARDS H. C., TROTT C. R.: Kokkos: Enabling performance portability across manycore architectures. In *2013 Extreme Scaling Workshop (xsw 2013)* (2013), IEEE, pp. 18–24. 21

[Fis13] FISER M.: Real time visualization of 3d vector field with cuda, 2013. URL: http://www.marekfiser.com/Projects/Real-time-visualization-of-3D-vector-field-with-CUDA. 6

[GHS*14] GUO H., HONG F., SHU Q., ZHANG J., HUANG J., YUAN X.: Scalable lagrangian-based attribute space projection for multivariate unsteady flow data. In *2014 IEEE Pacific Visualization Symposium* (2014), IEEE, pp. 33–40. 19

[GJ10] GARTH C., JOY K. I.: Fast, memory-efficient cell location in unstructured grids for visualization. *IEEE Transactions on Visualization and Computer Graphics 16*, 6 (2010), 1541–1550. 11, 13, 14

[Gol12] GOLDSTINE H. H.: *A History of Numerical Analysis from the 16th through the 19th Century*, vol. 2. Springer Science & Business Media, 2012. 4

[GTS*04] GARTH C., TRICOCHE X., SALZBRUNN T., BOBACH T., SCHEUERMANN G.: Surface techniques for vortex visualization. In *VisSym* (2004), vol. 4, pp. 155–164. 19

[GYHZ13] GUO H., YUAN X., HUANG J., ZHU X.: Coupled ensemble flow line advection and analysis. *IEEE Transactions on Visualization and Computer Graphics 19*, 12 (2013), 2733–2742. 19

[GZL*14] GUO H., ZHANG J., LIU R., LIU L., YUAN X., HUANG J., MENG X., PAN J.: Advection-based sparse data management for visualizing unsteady flow. *IEEE transactions on visualization and computer graphics 20*, 12 (2014), 2555–2564. 17, 20

[HK14] HORNUNG R. D., KEASLER J. A.: The raja portability layer: Overview and status. 21

[Hul92] HULTQUIST J. P. M.: Constructing stream surfaces in steady 3d vector fields. In *Proceedings of the 3rd Conference on Visualization '92* (Los Alamitos, CA, USA, 1992), VIS '92, IEEE Computer Society Press, pp. 171–178. 6

[HY00] HALLER G., YUAN G.: Lagrangian coherent structures and mixing in two-dimensional turbulence. *Physica D: Nonlinear Phenomena 147*, 3-4 (2000), 352–370. 6

[KKKW05] KRUGER J., KIPFER P., KONCLRATIEVA P., WESTERMANN R.: A particle system for interactive visualization of 3d flows. *IEEE Transactions on visualization and computer graphics 11*, 6 (2005), 744–756. 13, 14

[KL96] KENWRIGHT D. N., LANE D. A.: Interactive time-dependent particle tracing using tetrahedral decomposition. *IEEE Transactions on Visualization and Computer Graphics 2*, 2 (1996), 120–129. 11

[KPH*10] KASTEN J., PETZ C., HOTZ I., HEGE H.-C., NOACK B. R., TADMOR G.: Lagrangian feature extraction of the cylinder wake. *Physics of fluids 22*, 9 (2010), 091108. 7

[KSS05] KRUGER S. E., SCHNACK D. D., SOVINEC C. R.: Dynamics of the major disruption of a diii-d plasma. *Physics of Plasmas 12*, 5 (2005), 056113. 7

[KWA*11] KENDALL W., WANG J., ALLEN M., PETERKA T., HUANG J., ERICKSON D.: Simplified parallel domain traversal. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis* (2011), ACM, p. 10. 19

[L*15] LARSEN M., ET AL.: Ray Tracing Within a Data Parallel Framework. In *IEEE Pacific Visualization Symposium* (2015), pp. 279–286. 23

[L*17] LI S., ET AL.: Achieving Portable Performance For Wavelet Compression Using Data Parallel Primitives. In *EuroGraphics Symposium on Parallel Graphics and Visualization (EGPGV)* (2017), pp. 73–81. 23

[L*18] LESSLEY B., ET AL.: DPP-PMRF: Rethinking Optimization for a Probabilistic Graphical Model Using Data-Parallel Primitives. In *IEEE Symposium on Large Data Analysis and Visualization (LDAV)* (2018), pp. 34–44. 23

[LA90] LÖHNER R., AMBROSIANO J.: A vectorized particle tracer for unstructured grids. *Journal of Computational Physics 91*, 1 (1990), 22–31. 10, 12

[LBMC16] LESSLEY B., BINYAHIB R., MAYNARD R., CHILDS H.: External facelist calculation with data-parallel primitives. In *EuroGraphics Symposium on Parallel Graphics and Visualization (EGPGV)* (2016), pp. 11–20. 23, 24

[LGZY16] LIU R., GUO H., ZHANG J., YUAN X.: Comparative visualization of vector field ensembles based on longest common subsequence. In *2016 IEEE Pacific Visualization Symposium (PacificVis)* (2016), IEEE, pp. 96–103. 19

[LHK*16] LARSEN M., HARRISON C., KRESS J., PUGMIRE D., MEREDITH J. S., CHILDS H.: Performance modeling of in situ rendering. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (2016), SC '16, IEEE Press. 8

[LLN*15] LARSEN M., LABASAN S., NAVRÁTIL P. A., MEREDITH J. S., CHILDS H.: Volume Rendering Via Data-Parallel Primitives. In *EuroGraphics Symposium on Parallel Graphics and Visualization (EGPGV)* (2015), pp. 53–62. 23, 24

[LPM*17] LESSLEY B., PERCIANO T., MATHAI M., CHILDS H., BETHEL E. W.: Maximal clique enumeration with data-parallel primitives. In *IEEE Symposium on Large Data Analysis and Visualization (LDAV)* (2017), pp. 16–25. 23

[LSP14] LU K., SHEN H.-W., PETERKA T.: Scalable computation of stream surfaces on large scale vector fields. In *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (2014), IEEE, pp. 1008–1019. 18, 19, 20

[MCHG13] MÜLLER C., CAMP D., HENTSCHEL B., GARTH C.: Distributed parallel particle advection using work requesting. In *2013 IEEE Symposium on Large-Scale Data Analysis and Visualization (LDAV)* (2013), IEEE, pp. 1–6. 19, 20

[MMA*13] MAYNARD R., MORELAND K., ATYACHIT U., GEVECI B., MA K.-L.: Optimizing Threshold for Extreme Scale Analysis. In *Visualization and Data Analysis (VDA)* (2013), vol. 8654, International Society for Optics and Photonics, p. 86540Y. 23

[MSU*16] MORELAND K., SEWELL C., USHER W., LO L.-T., MEREDITH J., PUGMIRE D., KRESS J., SCHROOTS H., MA K.-L., CHILDS H., ET AL.: Vtk-m: Accelerating the visualization toolkit for massively threaded architectures. *IEEE computer graphics and applications 36*, 3 (2016), 48–58. 9, 21

[Nic07] NICKOLLS J.: Gpu parallel computing architecture and cuda programming model. In *2007 IEEE Hot Chips 19 Symposium (HCS)* (2007), IEEE, pp. 1–12. 13

[NLL*12] NOUANESENGSY B., LEE T.-Y., LU K., SHEN H.-W., PETERKA T.: Parallel particle advection and FTLE computation for time-varying flow fields. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (2012), IEEE Computer Society Press, p. 61. 17, 20, 21

[NLS11] NOUANESENGSY B., LEE T.-Y., SHEN H.-W.: Load-balanced parallel streamline generation on large scale vector fields. *IEEE Transactions on Visualization and Computer Graphics 17*, 12 (2011), 1785–1794. 17, 20

[P*10] PARKER S. G., ET AL.: OptiX: A General Purpose Ray Tracing Engine. *ACM Transactions on Graphics (TOG) 29*, 4 (2010), 1–13. 22

[P*18] PUGMIRE D., ET AL.: Performance-Portable Particle Advection with VTK-m. In *EuroGraphics Symposium on Parallel Graphics and Visualization (EGPGV)* (2018), pp. 45–55. 23, 24

[PCG*09] PUGMIRE D., CHILDS H., GARTH C., AHERN S., WEBER G. H.: Scalable computation of streamlines on very large datasets. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis* (2009), ACM, p. 16. 19, 20

[PHC*20] PERCIANO T., HEINEMANN C., CAMP D., LESSLEY B., BETHEL E. W.: Shared-Memory Parallel Probabilistic Graphical Modeling Optimization: Comparison of Threads, OpenMP, and Data-Parallel Primitives. In *International Conference on High Performance Computing* (2020), pp. 127–145. 22, 23

[Pop03] POPINET S.: Gerris: a tree-based adaptive solver for the incompressible euler equations in complex geometries. *Journal of Computational Physics 190*, 2 (2003), 572–600. 5

[PRN*11] PETERKA T., ROSS R., NOUANESENGSY B., LEE T.-Y., SHEN H.-W., KENDALL W., HUANG J.: A study of parallel particle tracing for steady-state and time-varying flow fields. In *2011 IEEE International Parallel & Distributed Processing Symposium* (2011), IEEE, pp. 580–591. 17, 19, 20

[PYK*18] PUGMIRE D., YENPURE A., KIM M., KRESS J., MAYNARD R., CHILDS H., HENTSCHEL B.: Performance-Portable Particle Advection with VTK-m. In *Eurographics Symposium on Parallel Graphics and Visualization* (2018), Childs H., Cucchietti F., (Eds.), The Eurographics Association. doi:10.2312/pgv.20181094. 14, 15

[RTBS05] RHODES P. J., TANG X., BERGERON R. D., SPARR T. M.: Iteration aware prefetching for large multidimensional scientific datasets. In *Proc. of the 17th international conference on Scientific and statistical database management (SSDBM)* (2005), pp. 45–54. 17

[SBK06] SCHIRSKI M., BISCHOF C., KUHLEN T.: Interactive particle tracing on tetrahedral grids using the gpu. In *Proceedings of Vision, Modeling, and Visualization (VMV)* (2006), pp. 153–160. 11, 13, 14

[SH96] SUJUDI D., HAIMES R.: Integration of particles and streamlines in a spatially-decomposed computation. *Proceedings of Parallel Computational Fluid Dynamics, Los Alamitos, CA* (1996). 16

[Ske93] SKEEL R. D.: Variable step size destabilizes the störmer/leapfrog/verlet method. *BIT Numerical Mathematics 33*, 1 (1993), 172–175. 4

[SLH*15] SEWELL C., LO L.-T., HEITMANN K., HABIB S., AHRENS J.: Utilizing many-core accelerators for halo and center finding within a cosmology simulation. In *2015 IEEE 5th Symposium on Large Data Analysis and Visualization (LDAV)* (2015), IEEE, pp. 91–98. 23

[SML96] SCHROEDER W. J., MARTIN K. M., LORENSEN W. E.: The Design and Implementation of an Object-Oriented Toolkit for 3D Graphics and Visualization. In *Proceedings of the 7th conference on Visualization* (1996), IEEE Computer Society Press, pp. 93–ff. 22

[ST19] SHIINA S., TAURA K.: Almost deterministic work stealing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (New York, NY, USA, 2019), SC '19, ACM, pp. 47:1–47:16. 19

[SVWHP94] SADARJOEN I. A., VAN WALSUM T., HIM A. J., POST F. H.: *Particle tracing algorithms for 3D curvilinear grids.* Delft University of Technology, 1994. 11

[top20] Top 500, 2020. URL: https://www.top500.org/lists/top500/2020/11. 13

[USM96] UENG S.-K., SIKORSKI C., MA K.-L.: Efficient streamline, streamribbon, and streamtube constructions on unstructured grids. *IEEE Transactions on Visualization and Computer Graphics 2*, 2 (1996), 100–110. 11, 12

[VKP00] VERMA V., KAO D., PANG A.: A flow-guided streamline seeding strategy. In *Proceedings Visualization 2000. VIS 2000 (Cat. No. 00CH37145)* (2000), IEEE, pp. 163–170. 6

[W*14] WALD I., ET AL.: Embree: A Kernel Framework for Efficient CPU Ray Tracing. *ACM Transactions on Graphics (TOG) 33*, 4 (2014), 1–8. 22

[YCM19] YENPURE A., CHILDS H., MORELAND K.: Efficient Point Merging Using Data Parallel Techniques. In *EuroGraphics Symposium on Parallel Graphics and Visualization (EGPGV)* (2019), pp. 79–88. 23, 24

[YWM07] YU H., WANG C., MA K.-L.: Parallel hierarchical visualization of large time-varying 3d vector fields. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing* (2007), ACM, p. 24. 17, 20

[ZGH*17] ZHANG J., GUO H., HONG F., YUAN X., PETERKA T.: Dynamic load balancing based on constrained kd tree decomposition for parallel particle tracing. *IEEE transactions on visualization and computer graphics 24*, 1 (2017), 954–963. 17, 20

[ZGY16] ZHANG J., GUO H., YUAN X.: Efficient unsteady flow visualization with high-order access dependencies. In *2016 IEEE Pacific Visualization Symposium (PacificVis)* (2016), IEEE, pp. 80–87. 17, 20