DOI: 10.1111/cgf.14858
EUROVIS 2023
S. Bruckner, R. Raidou, and C. Turkay
(Guest Editors)

COMPUTER GRAPHICS *forum*
Volume 42 (2023), Number 3
STAR – State of The Art Report

State-of-the-Art Report on Optimizing Particle Advection Performance

A. Yenpure[1,6] S. Sane[2] R. Binyahib[3] D. Pugmire[4] C. Garth[5] H. Childs[1]

[1]University of Oregon, USA, [2]Luminary Cloud Inc., USA, [3]Intel Inc., USA
[4]Oak Ridge National Laboratory, USA, [5]University of Kaiserslautern, Germany, [6]Kitware Inc., USA

Abstract

The computational work to perform particle advection-based flow visualization techniques varies based on many factors, including number of particles, duration, and mesh type. In many cases, the total work is significant, and total execution time ("performance") is a critical issue. This state-of-the-art report considers existing optimizations for particle advection, using two high-level categories: algorithmic optimizations and hardware efficiency. The sub-categories for algorithmic optimizations include solvers, cell locators, I/O efficiency, and precomputation, while the sub-categories for hardware efficiency all involve parallelism: shared-memory, distributed-memory, and hybrid. Finally, this STAR concludes by identifying current gaps in our understanding of particle advection performance and its optimizations.

1. Introduction

Flow visualization techniques are used to understand flow patterns and movement of fluids in many fields, including oceanography, aerodynamics, and electromagnetics. Many flow visualization techniques operate by placing massless particles at seed locations, displacing those particles according to a vector field to form trajectories, and then using those trajectories to create a renderable output. Each trajectory is calculated via a series of "advection steps," where each step advances a particle a short distance by solving an ordinary differential equation.

Particle advection workloads can be quite diverse across different flow visualization algorithms and grid types. The size of the workload varies based on many factors, including the number of particles, the duration of advection, the velocity field evaluation, and any analysis associated with each advection step. One crucial factor affecting performance is the total number of advection steps, which depends on both the number of particles and their duration of advection. Flow visualization techniques vary in terms of particle count and duration, with some techniques utilizing numerous particles that go for short durations and others using few particles that go for longer periods. In certain cases, such as studying ocean flow [OPF*12], many particles may be required for extended periods, resulting in billions of advection steps or more. With respect to the velocity field evaluation, uniform grids require only a few operations, while unstructured grids require many more (for cell location and interpolation). In all, the diverse nature of particle advection workloads means there are diverse approaches for optimizing performance.

The main goal of this state-of-the-report is to survey the space of optimizations for particle advection. After providing a background on the building blocks for particle advection in Section 2, the survey describes two high-level categories. First, Section 3 surveys algorithmic optimizations. Then, Section 4 surveys approaches for utilizing hardware more efficiently, with nearly all of these works utilizing parallelism. Contrasting the two high-level categories, Section 3 is about reducing the amount of work to perform, while Section 4 is about executing a fixed amount of work more quickly. In terms of how to read this survey, Sections 3 and 4 can be read in any order. In particular, readers interested in a particular algorithmic optimization or technique for hardware efficiency can skip to the corresponding subsection. That said, readers new to this topic should start with Section 2 to gain a basic understanding of particle advection performance.

With respect to previously published literature, this survey is the first effort to provide a state-of-the-art report for optimizing particle advection performance. The closest work to our own is the survey on distributed-memory parallel particle advection by Zhang and Yuan [ZY18]. There are two main ways our survey is distinct and novel in its contribution. First, our survey considers a broader context, i.e., it considers algorithmic optimizations and additional types of parallelism. Second, our discussion of distributed-memory techniques includes a new summarization of workloads and parallel characteristics (specifically Table 5), as well as, includes recent works appearing since their publication. Further, there have been many other excellent surveys involving flow visualization and particle advection: feature extraction and tracking [PVH*03], dense and texture-based techniques [LHD*04], topology-based flow techniques [LHZP07] and a subsequent survey focusing on topology for unsteady flow [PPF*11], integration-based, geometric flow vi-

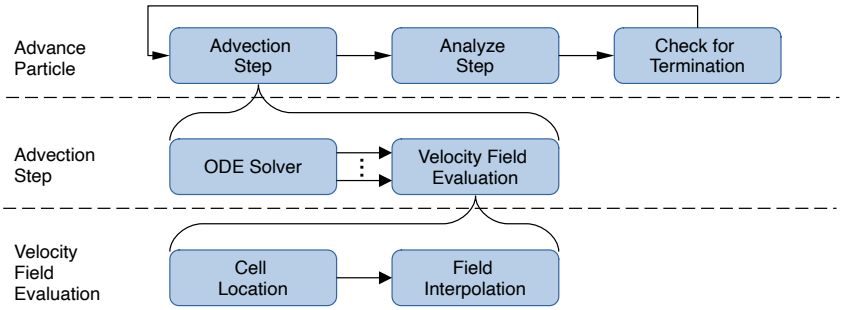


Figure 1: *Organization of the components for a particle advection-based flow visualization algorithm. The components are arranged in three rows in decreasing levels of granularity from top to bottom. In other words, the components at the bottom are building blocks for the components at higher levels. The top row shows components that define the movement and analysis of a particle. The loop in the top row indicates its components are executed repeatedly until the particle is terminated. The middle row shows components that define a single step of advection. The 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). The frequently-used Runge-Kutta 4 ODE solver requires four such velocity field evaluations. Finally, as depicted in the bottom row, 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.*

sualization [MLP*10], and seed placement and streamline selection [SBGC20]. This STAR complements these existing surveys — while some of these works consider aspects of performance within their individual focal point, none of the surveys focus on optimizing particle advection performance.

2. Particle Advection Background

Flow visualization algorithms perform three general operations:

- ***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 three operations often happen in sequence, but in some instances they happen in an overlapping fashion (i.e., seed, advance, seed more, advance more, etc.)

Our organization, which is illustrated in Figure 1, focuses on the "advance particles" portion of flow visualization algorithms. It divides the components into three levels of granularity.

The "top" level of our organization considers the process of advancing a single particle. It is divided into three components:

- ***Advection Step:*** advances a particle from its current location to its next location.
- ***Analyze Step:*** analyzes the advection step taken. The specifics of the analysis vary by flow visualization algorithm, and could be as simple as storing the particle's new location in memory (e.g.., streamlines) or could involve more computation (e.g., calculating statistics, checking if a particle entered a critical region, etc.).
- ***Check for Termination:*** determines whether a particle should be terminated. Similar to the Analyze Step component, flow visualization algorithms define specific criteria for when to terminate a particle, i.e., number of steps, distance traveled, time elapsed, etc.

The process of advancing a particle involves applying these three components iteratively until the termination criteria are reached.

The "middle" level of our organization considers the process of completing a single advection step for a particle. This level has two components:

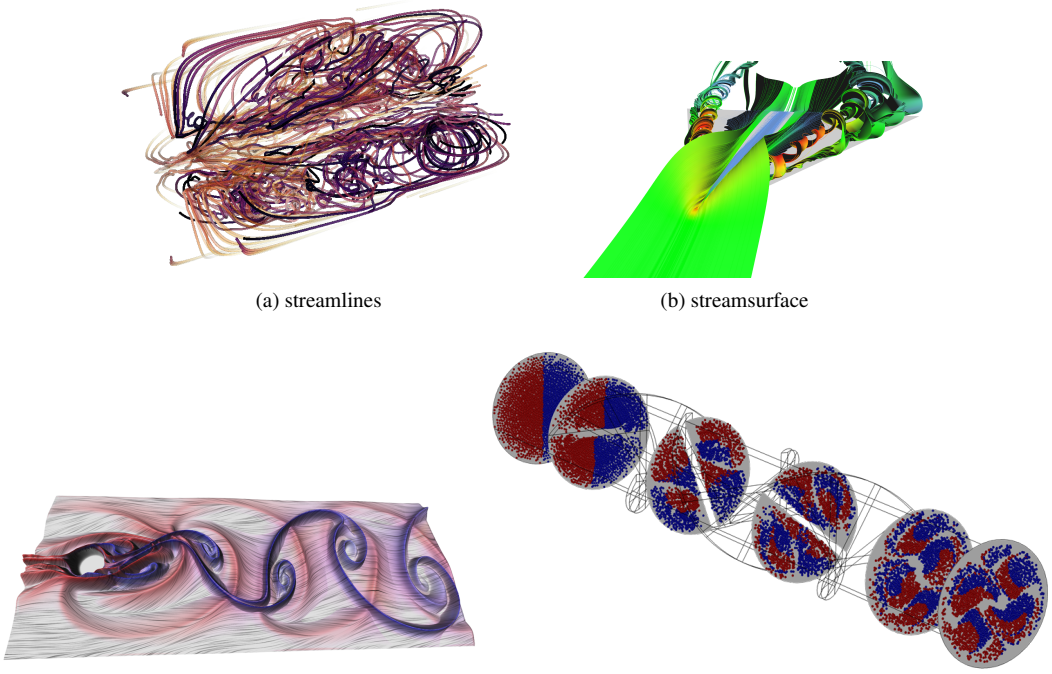
- ***ODE Solver***: calculates a particle's displacement to a new position by solving an ordinary differential equation (ODE).
- ***Velocity Field Evaluation:*** calculates the velocity value at a specific location by interpolating within the located cell.

The "bottom" level of our organization considers the process of velocity field evaluation. This level also has two components:

- ***Cell Location:*** locates the cell that contains some location.
- ***Field Interpolation:*** calculates velocity field at a specific location via interpolation of surrounding velocity values.

Thus, to calculate the velocity value at some point P, a cell location process is first used to identify the cell C containing P, and then interpolation of the velocity field is performed to calculate the velocity at P using information at the vertices of C.

Flow visualization techniques use these components in different ways, resulting in varying performance across the algorithms. One way of comparing the different algorithms' performance can be the workload characteristics of the algorithms, which roughly translates to the total computation required by an algorithm. This workload can be defined as the total number of advection steps completed by the algorithm, which is the product of the total number of particles required by the algorithm and the number of steps expected to be completed by each particle. Figure 2 shows examples of four flow visualization algorithms that demonstrate significant



(a) streamlines (b) streamsurface

(c) FTLE (d) Poincaré

Figure 2: *Example flow visualizations from four representative algorithms. Subfigure (a) shows streamlines rendered over a slice of jet plume data created using the Gerris Flow Solver [Pop03], subfigure (b) shows a streamsurface which is split by turbulence and vortices that can be observed towards the end [Fis13], subfigure (c) shows attracting (blue) and repelling (red) Lagrangian structures extracted from ridges of the finite-time Lyapunov exponents (FTLE) of a von Korman vortex street simulation [KPH*10], and subfigure (d) shows a Poincaré plot of a species being dissolved in water, where the color of the dots represent the level of dissolution [Lex15].*

differences in their workloads and behaviors. Table 1 highlights the differences between the workloads for the example algorithms.

3. Algorithmic Optimizations

This section surveys algorithmic optimizations for the particle advection building blocks, i.e., techniques for executing a given workload using fewer operations. Some of the building blocks do not particularly lend themselves to algorithmic optimizations. For example, a RK4 solver requires a fixed number of FLOPS, and the only possible "optimization" would be to use a different solver or adaptive step sizes. 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. This section discusses four optimizations that address the algorithmic challenges. Section 3.1 discusses optimizations to ODE solvers, Section 3.2 discusses optimizations for cell location, Section 3.3 discusses strategies to improve I/O efficiency, and finally Section 3.4 discusses strategies that involve precomputation.

3.1. ODE Solvers

The fundamental problem underlying particle advection is solving of ODEs. Many methods are available for this, with different trade-offs, and a comprehensive review is beyond the scope of this work. We thus refer the reader to the excellent book by Hairer et al. [HNW00] for a more thorough overview. Due to the generally (numerically) benign nature of vector fields used in visualization, a set of standard schemes is used in many visualization implementations.

Beyond the Euler and the fourth-order Runge-Kutta (RK4) methods, techniques with adaptive step size control have proven useful. The primary objective of such methods is to allow precise control over the local error of the approximation of the solution, which is achieved by automated selection of the step size (which in turn controls the local approximation error) in each step. Often used methods in this context are the Runge-Kutta Fehlberg (RFK) method [HNW00], the Runge-Kutta Cash-Karp (RKCK) method [HNW00], and the Dormand-Prince fifth-order scheme (DOPRI5) [PD81]. Additional significant performance benefits can be realized if relaxed error requirements allow adaptive step sizing to perform fewer, larger steps, when compared to fixed-step size methods. This is the case in many visualization scenarios, where small-scale errors are reduced in further processing, e.g. in rendering or estimation of derived quantities. While the magnitude of such benefits depends on a variety of factors that are hard to quantify, using an adaptive step sizing method is generally recommended. Corresponding implementations are widely available, e.g. in the VTK framework [GSBW12, HMCA15].

Table 1: *Parameters for seeding strategy, the number of seeds, and the number of steps for four representative flow visualization algorithms. (a) describes the parameters and their classifications, and (b) presents the typical values for the four algorithms.*

Seeding Strategy	***Sparse***	***Packed***	***Seeding Curves***
Number of Seeds	***Small*** ≤1/1K cells	***Medium*** ~1/100 cells	***Large*** ≥1/cell
Number of Steps	***Small*** ≤100	***Medium*** ~1K	***Large*** ≥10K

(a) Each parameter is classified in three categories.

Algorithm	Seeding	# Seeds	# Steps
Streamlines	Sparse/Packed	Small	Large
Streamsurface	Seeding Curves	Small	Large
FTLE	Packed	Large	Small
Poincarè	Packed	Medium	Large

(b) Typical parameter configurations for four different flow visualization algorithms. That said, individual flow visualization algorithms, including the four listed here, can have significant variation in their parameter configurations across contexts, to the point of having different values for each category.

For specialized applications, substantial performance benefits may be obtainable by relying on domain-specific integration schemes that generally exhibit higher accuracy orders and thus allow larger step sizes than general-purpose schemes. For example, Sanderson et al. [SCT*10] report substantial speedup from employing an Adams-type scheme for visualizing high-order fusion simulation data. However, general guidance on the selection of optimal schemes for domain-specific vector field data remains elusive.

3.2. Cell Locators

Evaluating the velocity field at a specific location in discrete data requires interpolation. Typically, the entire velocity field domain is divided into a set of cells of same or different types. The velocity values given at the vertices of the cell containing the location are then interpolated using pre-defined schemes, e.g. schemes used in finite-element literature [Ern21] or in geometric modeling [Far02]. For grids whose cells exhibit an implicit structure, such as uniform or rectilinear grids, it is easy to identify the relevant cell; however, for other grid types (cf. e.g. [SML97]), *cell location* refers to the problem of quickly locating the cell containing the location. Note that particle advection does not fundamentally rely on cell location. For example, given specialized data representations, interpolation may be performed in a completely different manner [SCT*10].

Since it is difficult to derive a general approach from such special cases, here, we consider the more ubiquitous case of interpolation in piecewise (multi-)linear (un-)structured grids, for which cell locators are essentially required to achieve acceptable particle advection performance. Cell locators rely on auxiliary data structures that partition candidate cells spatially, and are typically constructed in a pre-processing step. They induce a linear memory overhead in the number of cells N, while generally reducing the computational complexity of queries. Hence, many cell location schemes allow trading off memory overhead for improved performance. A variety of schemes have been developed for different scenarios. For example, limited available memory, e.g. on GPUs, can be addressed through multi-level data structures.

Note that variants of the cell location problem have been considered in visualization contexts other than particle advection, e.g. in volume rendering [Kno06], and for more general problems [com08]. In the following however, we limit our consideration of techniques to those described specifically in conjunction with particle advection. According to Lohner and Ambrosiano [LA90] the process of cell location can follow one of the following three approaches.

Using a Cartesian background grid: Cells are spatially subdivided using a superimposed Cartesian grid, storing a list of overlapping cells of the original grid per superimposed cell. The superimposed cell can be found in constant time, and cell location then requires traversing all overlapping cells to find the actual containing cell for the query point. While conceptually simple, this approach is not ideal if the background grid exhibits large variances in cell sizes, either incurring excessive storage overhead or decreased performance, depending on the resolution of the superimposed grid.

Using tree structures: Tree-based approaches leverage the principle of n-ary search trees to achieve cell location complexity in $\mathcal{O}(\log N)$. A basic approach to hierarchical cell location is the use of octrees [SML97, WvG92]. Each leaf of an octree stores cells whose bounding box overlaps with the leaf extents. Leaves are subdivided until either a maximum depth is reached, or the number of overlapping cells falls below an upper bound. Cell location proceeds by traversing the octree from the root and descending through nodes until a leaf is reached, which then contains all the candidate cells. Due to the regular nature of octree subdivision, this approach does not work well with non-uniform vertex distributions, requiring either too many levels of subdivision and thus a considerable memory overhead, or does not shrink the candidate cell range down to acceptable levels.

Using kd-trees instead of octrees facilitates non-uniform subdivision, at the cost of generally deeper trees and a storage overhead. An innovative approach was given by Langbein et al. [LST03], based on a kd-tree storing just the vertices of an unstructured grid. This allows quick location of a grid vertex close to the query point; using cell adjacency, ray marching is used to traverse the grid towards the query point using cell walking. Through clever storage of the cell-vertex incidence information, storage overhead can be kept reasonable.

Garth and Joy described the *cell tree* [GJ10], which employs a kd-tree-like bounding interval hierarchy based on cell bounding boxes to quickly identify candidate cells. This allows a flexible trade-off between performance and storage overhead and allows rapid cell location even for very large unstructured grids with hundreds of millions of cells on commodity hardware and on memory-limited GPU architectures.

Addressing storage overhead directly, Andrysco and Tric-

oche [AT10] presented an efficient storage scheme for kd-trees and octrees, based on *compressed sparse row* (CSR) storage of tree levels, termed *Matrix *Trees*. The tree data structure is encoded as a sparse matrix in CSR representation. This alleviates most of the memory overhead of kd-trees, and they are able to perform cell location with reduced time and space complexity when compared with typical tree data structures.

Overall, non-uniform hierarchical subdivision can accommodate large meshes with significant variations in cell shapes and sizes well. While Lohner and Ambrosiano [LA90] note that vectorization of this approach is challenging as tree-based schemes introduce additional indirect addressing, vectorization is still possible on modern CPU and GPU architectures with good performance [GJ10].

Using successive neighbor searches: For the case of particle integration, successive interpolation queries exhibit strong coherence and are typically spatially close. This enables a form of locality caching: For each interpolation query except the first, the cell that contained the previous query point is checked first. If it does not contain the interpolation point, its immediate neighbors are likely to contain it, potentially reducing the number of cells to check. The initial interpolation point can be located using a separate scheme, e.g. as discussed above.

Lohner and Ambrosiano [LA90], as well as Ueng et al. [USM96], adopted a corresponding 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, requiring a pre-decomposition for general unstructured grids. Note that when applied to tetrahedral meshes, the successive neighbor search approach is sometimes also referred to as ***tetrahedral walk*** [SBK06, BRKE*11].

Kenwright and Lane [KL96] extended the work by Ueng et al. by improving the technique to identify the tetrahedron that contains a particle. Their approach uses fewer floating point operations for cell location compared to Ueng et al.

Successive neighbor search is also naturally incorporated in the method of Langbein et al. [LST03]; ray casting with adjacency walking begins at the previous interpolation point in this case.

3.3. I/O Efficiency

Simulations with very large numbers of cells often output their vector fields in 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 is known as out-of-core computation. One of the significant bottlenecks for flow visualization algorithms while performing out-of-core computations is the I/O cost. 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 efficiency of I/O operations.

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 graph's nodes represent the data blocks, and the edges are weighted based on the probability that a particle travels from one block to another. The information from the graph is used during runtime to minimize data block misses. Their method demonstrated speed-ups over the Hilbert curve layout [Hil91]. Chen et al. [CNLS12] extended the previous work to unsteady vector fields for out-of-core computation of pathlines. Their results show a performance improvement of 10%-40% compared to the Z-curve layout [ZZ94, ZZO03]. Further, Chen et al. [CS13] extended their work by introducing a seed scheduling strategy to be used alongside the graph-based data layout. They demonstrated an efficient out-of-core approach for calculating FTLE by minimizing the number of data blocks loaded for flow map generation. This involved estimating the total data block accesses given the seed assignment for each round and the path information calculated from the graph model. They used this estimate to optimize the grouping of seeds and the order of execution. The performance improvements observed against the Z-curve layout ranged from 8% to 32%.

3.4. Precomputation

Besides optimizing individual particle advection building blocks, optimization of certain flow visualization workloads can benefit from a two-stage approach. During the first stage, a set of particle trajectories can be computed to inform data access patterns, or serve as a basis for interpolating new trajectories. Notably, as demonstrated by recent work, these trajectories can be used to train deep learning models as well. When used to inform data access patterns, the accuracy of trajectories derived in the second stage is not impacted by the precomputed trajectories. Whereas, for methods using precomputed trajectories as a basis for interpolating new trajectories, the resolution and spatiotemporal sampling strategies used impact reconstruction accuracy. Thus, depending on the objectives, the number of trajectories precomputed during the first stage varies.

Precomputed trajectories can inform data access patterns to provide a strategy to improve I/O efficiency, as mentioned in the context of the study by Chen et al. [CXLS11] in the previous section. A similar approach was studied by Nouansengsy et al. [NLS11] to improve load balancing in a distributed memory setting. In these cases, the first stage is a preprocessing step and a small number of particles might be advected to form the set of precomputed trajectories. Hong et al. [HZY18] proposed a novel method to learn the access patterns for data blocks in unsteady flow fields using a deep learning technique "Large Short Term Memory (LSTM)," as they are much better for representing higher-order, long-term dependencies. They trained their approach while using a few pathlines as a means to train the LSTM model by generating a sequence of block transitions from the initial blocks for the particles. The trained models are later employed as a block prediction and prefetching mechanism. The study showed improved performance in comparison to not using any prefetching, employing the CPU for particle tracing and the GPU to execute the model.

For a computationally expensive particle advection workload, a strategy to accelerate the computation or improve interactivity

Table 2: *Summary of studies considering algorithmic optimizations to particle advection. Studies that do not report quantitative performance improvements are not mentioned in the table. The asterisk for entries in the data size column represent unstructured grids.*

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

of unsteady flow visualization is to divide the workload into two sets. The first set includes basis particle trajectories computed using high-order numerical integration. The second set includes particle trajectories that are derived by interpolating the precomputed basis trajectories from the first set. If new particle trajectories can be derived, while maintaining accuracy requirements, from the precomputed set faster than numerical integration, then the total computational cost of the workload can be reduced.

Hlawatsch et al. [HSW10] introduced a hierarchical scheme to construct integral curves, streamlines or pathlines, using sets of precomputed short flow maps. They demonstrated the approach for the computation of FTLE and line integral convolution. Although the method introduces reduced accuracy, they demonstrate their approach can result in an order of magnitude speed up for long integration times.

To accelerate the computation of streamline workloads, Bleile et al. [BSGC17] employed block exterior flow maps (BEFMs) produced using precomputed trajectories. BEFMs, i.e., a mapping of block-specific particle entry to exit locations, are generated to map the transport of particles across entire blocks in a single interpolation step. Thus, when a new particle enters a block, instead of performing an unknown number of numerical integration steps to traverse the region within the block, based on the mapping information provided by precomputed trajectories, the location of the particle exiting (or terminating within) the block can be directly interpolated as a single step. Depending on the nature of the workload, large speedups can be observed using this strategy. For example, Bleile et al. [BSGC17] observed up to 20X speed up for a small loss of accuracy due to interpolation error.

To support exploratory visualization of time-varying vector fields, Agranovsky et al. [ACG*14] proposed usage of in situ processing to extract accurate Lagrangian representations. In the context of large-scale vector field data, and subsequent temporally sparse settings during post hoc analysis, reduced Lagrangian representations offer improved accuracy-storage propositions compared to traditional Eulerian approaches, as well as, supporting acceleration of trajectory computation during post hoc analysis. By seeding the precomputed trajectories along a uniform grid, structured (uniform or rectilinear) grid interpolation performance can be achieved during post hoc analysis. To further optimize the accuracy of reconstructed pathlines in settings of temporal sparsity, research has considered how varying the set of precomputed trajectories can improve performance and accuracy-storage propositions. For example, the use of adaptive sampling [BT13], longer precomputed trajectories [SCB19], statistical sampling [RPD19], rank-specific local flow maps [SYB*21], and improved search structures for interpolation [COJ15] have been studied. Unstructured sampling strategies, however, could increase the cost when using standard post hoc interpolation techniques and diminish computational performance benefits.

More recently, Han et al. [HSJ22] used precomputed particle trajectories to train a deep learning model and replace interpolation with an inference process to derive new particle trajectories in unsteady flow. Once trained, the model has a reduced memory footprint and can be used to infer several locations along thousands of pathlines in a few seconds using a GPU. While the study demonstrates promise using a 2D analytical dataset, the application of deep learning is yet to scale to accurately analyze large and complex unsteady flow.

3.5. Summary

Table 2 summarizes studies that address algorithmic optimizations and report performance improvements against a baseline implementation. The studies mentioned in the table either target opti-

mizations for cell location or perform better I/O operations. For ODE solvers and precomputation, reporting performance improvements is difficult because of an associated accuracy trade-off for better performance. 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 [LA90] achieved the speed of 14X. However, the other study [USM96] demonstrated a speed-up of around 1.6X. The works by Chen et al. [CXLS11, CNLS12, CS13] for efficient I/O for particle advection all demonstrated speed-ups up to 1.3X.

4. Using Hardware Efficiently

Flow visualization algorithms often share resources with large simulation codes, or require large amounts of computational resources of their own depending on the needs of the analysis task. As a consequence, flow visualization algorithms are often required to execute on supercomputers. Executing codes on supercomputers is expensive, and it is necessary that all analysis and visualization tasks execute with utmost efficiency. Modern supercomputers have multiple ways to make algorithms execute fast. Typically, supercomputers have thousands of nodes over which computation can be distributed, and each node has multi-core CPUs alongside multiple accelerators (e.g., GPUs) for parallelization. As a result, algorithms are expected to make efficient use of the available concurrency. This section discusses research for particle advection that addresses efficient usage of available hardware. Section 4.1 discusses research that aims to improve shared-memory (on-node) parallelism. Section 4.2 discusses research that aims to improve distributed memory parallelism. Section 4.3 discusses research that uses both shared and distributed memory parallelism.

4.1. Shared Memory Parallelism for Particle Advection

Shared memory parallelism refers to using parallel resources on a single node. 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 shared memory concurrency, and 2) performance portability. 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 sub-section discusses particle advection using shared memory parallelism in two parts. Section 4.1.1 discuss works to optimize the performance particle advection on GPUs, while Section 4.1.2 discusses works that use CPUs for improving the performance of particle advection.

4.1.1. Shared Memory Parallelism on GPUs

Most solutions that consider shared memory optimization focus on improving performance using GPUs, which are particularly beneficial when advecting numerous particles. Since particles can be advected independently from one another, each particle can be scheduled with a separate thread on the GPU, maximizing the available concurrency. That said, the studies that have attempted to address performance issues related to particle advection using GPUs have employed different ways of utilizing the GPU's capabilities.

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 GPU. 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 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.

We organize the rest of this section into two parts: approaches for specific applications and cell locators.

4.1.1.1. GPU Approaches for Specific Applications This section considers works that use GPUs to solve problems for three distinct areas: feature detection, uncertainty visualizations, and Line Integral Convolution (LIC).

Algorithms designed to detect and analyze fluid flow features,

such as turbulence and critical points, necessitate the processing of vast numbers of particles, which leads to substantial computational demands. These algorithms also generate copious amounts of data output, making it crucial to carefully consider their implementation on GPUs due to memory limitations. Ghaffari et al. [GGW22] developed an interactive visualization system for analyzing turbulent flow superstructures, utilizing data compression for efficient GPU streaming and on-the-fly decompression for particle advection and feature detection tasks. Günther et al. [GKT16] created a Monte Carlo-based method for visualizing Lagrangian fields in fluid flows, demonstrating GPUs speedups of an order of magnitude better than its multi-threaded CPU counterpart, and later introduced an enhanced version [RGG20] using GPU shaders achieving 8X their initial research speed.

In many real-world scenarios, the vector field data for particle advection contains uncertainty. This uncertainty can arise from a variety of factors, such as simulation models and parameters, ensemble data, and errors introduced by temporal and/or spatial reduction. Moreover, the solvers used for particle advection, along with their parameters (e.g., tolerance, step size, etc.), can result in uncertainty regarding the computed particle trajectories. The computational costs of flow visualization in uncertain data can be high, particularly when extensive sampling is required. In this context, utilizing GPUs for high-performance solutions can be crucial. Rapp et al. [RD20] used GPUs to accelerate the visualization of transport barriers in stochastic flows. Guo et al. [GHP*16] used GPUs to accelerate the most computationally expensive kernels used for FTLE and LCS. Guo et al. [GHS*19] used an approach using both distributed memory parallelism and hybrid shared memory parallelism (CPU and GPU co processing) to estimate stochastic flow maps for analysis of uncertainty in unsteady flows. Preuss et al. [PWK21] used GPUs to accelerate a discrete probabilistic framework for dense, textured based flow visualization.

Liu et al. [LM05] presented a GPU-based implementation for unsteady flow line integral convolution (UFLIC), termed Accelerated ULID (AUFLIC). Their approach relied on reusing pathline computations such that only one particle is associated with one pixel, resulting in near real-time visualizations with an order of magnitude speed-up over a typical UFLIC algorithm. More recently, Ding et al. [DLYC15] proposed an implementation inspired by AUFLIC, however, their entire pipeline, including particle management, value scattering and depositing, post-processing is CUDA-accelerated, resulting inreal-time dense visualization of unsteady flows with high spatialtemporal coherence

4.1.1.2. Cell Locators for GPUs Bußler et al. [BRKE*11] presented a GPU-based tetrahedral walk for particle advection. Their approach for cell location borrowed heavily from the work by Schiriski et al. [SBK06] discussed in Section 3.2. However, they could 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 concluded 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 previously discussed in Section 3.2 that 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. They obtained good performance on GPUs despite no GPU-specific optimizations.

4.1.2. Shared Memory Parallelism on CPUs

Lane [Lan95] presented seminal work for parallelizing particle advection in a shared memory setting using CPUs. They evaluated the strong scaling of their parallelization on three different systems, using up to 8 CPUs, and achieved speedups of up to 3.6X.

Hentschel et al. [HGK*15] presented a solution focused on optimizing particle advection on CPUs. Their solution studied the performance benefits of using SIMD extensions on CPUs to achieve better performance. This paper addresses the general tendency of particles to move around in the flow field. This decreases memory locality of the data required to perform the advection computation. The study demonstrated the advantage of packaging particles into spatially local groups where SIMD extensions are able to be more efficient. The approach resulted in performance improvements of up to 5.6X over the baseline implementation.

Finally, Pugmire et al. [PYK*18] provided a platform portable solution for particle advection using the VTK-m [MSU*16] 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.

4.1.3. Summary

Table 3 presents an overview of published research on 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 3 shows a preliminary result for understanding the characteristics of shared memory parallel particle advection based on previous findings form Pugmire et al. [PYK*18] and from newer experiments designed for the purpose of this report. Some of the key observations are:

1. Workloads with larger numbers of particles scale better with added parallelism for the same amount of total work – this is expected as more particles lead to more concurrency.
2. The studies that used the RK4 integrator generally scaled better than the ones that used the Euler integrator – this behavior can be generalized to other higher order solvers as well as they perform significantly more arithmetic work per memory access.
3. Experiments with unstructured data scaled better on the CPU than on the GPU – this can be explained by 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

Table 3: *Summary of studies considering optimizations for shared memory particle advection. The asterisk for entries in the data size column represent unstructured grids.*

Algorithm	Application	Intent / Evaluation	Data Size	Time Steps	Seed Count	Performance
Krüger et al. [KKKW05]	source-dest	Interactive flow visualization (steady) using GPUs	-	-	-	60 - 80X
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	16.5X
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 - 0.48X (GPUs) 0.29 - 0.36X (CPUs) 1.56 - 2.24X (GPUs) 0.79 - 0.84X (CPUs) 1.42 - 2.04X (GPUs) 0.51 - 0.59X (CPUs)

the P100 GPUs (Figure 3, 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 speedups achieved by two considered GPUs, where the P100 GPU is able to achieve speedups of over 125X and the K80 GPU achieves speedups of less than 12X. The performance difference of particle advection between two generations of GPUs can be significant. Existing studies fail to capture this relation, which makes it harder to estimate the speedup that can be realized. Understanding the performance characteristics of particle advection across different GPUs is a potential avenue for future work.

4.2. Distributed Memory Parallelism for Particle Advection

Fluid simulations are capable of producing meshes with billions or even trillions of cells. Analyzing and visualizing meshes this large to extract useful information demands significant resources, often close 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. Particle advection-based flow visualization algorithms often execute in a distributed memory setting. The objective of the distribution of work is to perform efficient computation, memory and I/O operations, 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 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. This method aims to reduce the cost of I/O operations, which is more expensive than the cost of performing computations.

Parallelize over particles: Particles are distributed among parallel processes 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.

Most distributed particle advection solutions are either an optimization of these two classes or a combination of them. The deci-

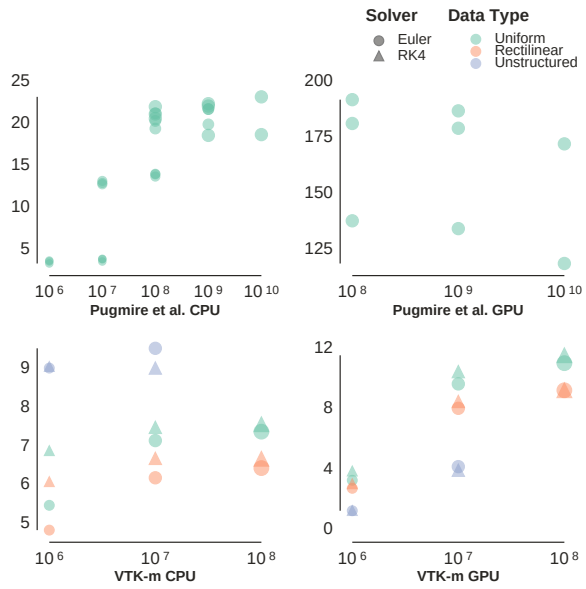


Figure 3: *Scatter plots for multi-core CPU and GPU speedups against serial experiments for particle advections workloads. The X axis represents the magnitude of the workload in terms of the total number of steps for each of the sub plots and the Y axis represents the speedup versus the corresponding serial experiment. 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 the study by Pugmire et al. [PYK*18], which 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, which used 12 CPU cores and a Nvidia K80 GPU.*

sion to choose between these two classes depends on multiple factors, of which Camp et al. [CGC*11] identify the most prominent to be:

Data set size: 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 can be more efficient.

Number of particles: Some flow visualization algorithms require small numbers of particles integrated over a long duration, while others require large numbers 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 owns 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 4.2.1 describes the optimizations for parallelizing distributed-memory particle advection in more depth. Section 4.2.2 summarizes findings from the survey of distributed particle advection studies.

4.2.1. Parallelization Methods

This section presents distributed particle advection works in three parts. Section 4.2.1.1 presents works that optimize parallelization over data. Section 4.2.1.2 presents works that optimize parallelization over particles. Section 4.2.1.3 presents works that use a combination of parallelization over data and particles.

Many of the works are implemented using the Message Passing Interface (MPI) [GLS99], which is the de facto standard for distributed-memory computations on supercomputers. Some works are "MPI-only," i.e., they only incorporate distributed-memory parallelism via MPI. Other works are "MPI-hybrid," i.e., they use both distributed-memory parallelism via MPI and also shared-memory parallelism (e.g., OpenMP, CUDA) within a compute node. Also, note there are two types of hybrid parallelism in this space, with one referring to what to parallelize over (particles, domains) and one referring to how to incorporate both distributed- and shared-memory parallelism. Finally, performance on supercomputers can vary greatly based on architecture, scale, message size, and other factors. That said, the MVAPICH project [PSCB21], an implementation of MPI, reported latency times to be between one microsecond to one hundred microseconds, based on message size [Pan19]. Of course, latency is just one aspect of the overall performance picture, which also includes marshaling and unmarshaling of messages, idle time while waiting for a message to be sent, and additional latencies within a compute node (i.e., CPU to GPU transfers).

4.2.1.1. *Parallelization over Data* Several solutions have been proposed to optimize the *"Parallelize over data"* method described above in Section 4.2. These solutions aimed to reduce the communication costs and maintain load balance.

Sujudi and Haimes [SH96] elicited the problems introduced by decomposing data into smaller blocks that can be used within the working memory of a single node. They presented important work in generating streamlines in a distributed memory setting using the parallelize over data 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 requests 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*11] 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 demonstrated performance improvements between 1.5X-6X 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, and 2) works that do not require knowledge of vector field.

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 performed 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 treated their spatiotemporal data as 4D data instead of considering the space and time dimensions as separate. They performed 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 were clustered hierarchically using a binary cluster tree based on the similarity of cells in a neighborhood. This hierarchical clustering helped them to partition data that ensured workload balance. It also enabled 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 an 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.

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. In static round-robin partitioning, data blocks are distributed among processes in a cyclic manner instead of a continuous manner. This partitioning method reduces the probability of assigning one processor a chunk of the data with a heavier workload (e.g., in the case of seed dense distribution). In dynamic geometric repartitioning, the data blocks are repartitioned during runtime depending on their workload. The algorithm checks the workload of each data block at regular intervals and redistributes the data blocks to assign similar workload across processes. The comparison of the two partitioning methods 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 the 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 from the study by Peterka et al. [PRN*11].

Zhang et al. [ZGH*17] proposed a method that is better at 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 demonstrated better load balance was achieved among the processes without the additional costs of pre-processing and expensive communication. Their results also demonstrate higher parallel efficiency. However, their work made 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.

Zhang [ZGYP18] et al. removed the pre-processing requirement from Nouanesengsy et al. [NLL*12] to support dynamic load balancing using data redistribution. As particles are traced, the amount of work done in each block are recorded. At regular user-specified intervals during the computation, this data is used to build a dependency graph of blocks traversed by particles. This dependency graph can be used to predict the future path of particles and is used to redistribute the data and particles to achieve better load balance. Their method improved the overall balance of work but comes at the cost of using synchronous communication and may require the movement of large amounts of data.

Sisneros et al. [SP16] studied the impact of communication granularity and the use of both synchronous and asynchronous communication. While the performance with respect to communication granularity is complex, the performance using asynchronous communication is significantly better than synchronous communication.

Schwartz et al. [SCP21] compared different machine learning approaches to formulate an oracle for optimizing particle advection. The oracle takes workload characteristics (number of particles, duration of advection, etc.) into consideration and determines the best execution settings for the algorithm. Their approach considered two techniques thoroughly, Neural Networks and Random Forests to optimize the oracle. By using the machine learning approaches, a maximum improvement of up to 20% was demonstrated.

Finally, Morozov et al. [MPG*21] presented an MPI-based approach for asynchronous communication and termination detection for parallel algorithms. They demonstrated that particle advection, which traditionally has been implemented as a bulk-synchronous parallel pattern, can be expressed as an asynchronous parallel operation. Each task only cares about the local computations for particles enqueued to it, terminating them when required, or enqueuing them to the other process's queue. Their approach resulted in speedups of up to 3.6X compared to the classical implementation.

In conclusion, pre-processing works can achieve load balance with an additional cost for *parallelize over data*. This cost increases for large volumes of data. The overall time for completing particle advection might not benefit from the additional cost of preprocessing, 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. [ZGH*17] is promising but still does not guarantee optimal load balancing.

4.2.1.2. *Parallelize over Particles* Previous works have explored different approaches to optimize the *"Parallelize over particles"* method described above in Section 4.2. Since the blocks of data are loaded whenever requested, the cost of I/O is a dominant factor in the total time [CPA*10]. 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 used 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.

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*11] 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 leveraged different memory hierarchies available on modern supercomputers to improve the performance of particle advection. The objective of the work is to reduce the 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 unless there is a priori knowledge of the data. 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 acquires work from the victim. In work-requesting, the thief requests work items, and the victim voluntarily shares it. In work-stealing, the thief directly accesses the victim's queue for work items without the victim knowing.

A large 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. Dinan et al. [DLS*09] demonstrated the scalability of the work-stealing approach. Lu et al. [LSP14] presented a technique for calculating stream surface efficiently using work-stealing. Their algorithm aimed 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 ad-

vancing 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. When a processor requires additional data to advance a segment, it requests the data from the processes that own the data block. Their solution demonstrated good load balancing and scalability.

Work stealing has been proven to be efficient in theory and practice. However, Dinan et al. reported its implementation is complicated.

Muller et al. [MCHG13] presented an approach that used work requesting for tracing particle trajectories. Their algorithm started 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 processes' need to load the data block for the particle. The results reported performance improvements between 30% to 60%.

While work-stealing and work requesting methods increase the communication overhead, the surveyed solutions show an improved overall performance and the majority of the time was spent in computations.

Binyahib et al. [BPC19] compared the parallelize over particle strategy to parallelize over data for its in-situ applicability. Their findings suggest that for workloads where particles are densely seeded in a certain region of the data, parallelize over particles is a much better strategy and can result in speedups up to 10X.

While solutions like data prefetching reduce I/O time, they 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.

Table 4: *Recommendation of parallelization strategy for particle advection workloads based on features of the problem. This table appears in the survey by Binyahib [Bin19].*

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

4.2.1.3. ***Hybrid Particle Advection*** 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 supervisor-worker model. The processes were divided into groups, and each group had a supervisor process. The supervisor 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 supervisor. The supervisor 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.

Kendall et al. [KWA*11] provided a hybrid solution which they call DStep and works like the MapReduce framework [DG08]. Their algorithm used groups for processes as well and has a supervisor 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 supervisor 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 supervisor processes. The algorithm provided an efficient and scalable solution for particle tracing and has been used by other works [GYHZ13, GHS*14, LGZY16].

Binyahib et al. [BPC21] proposed a new 'HyLiPoD' algorithm for particle advection. Their work was inspired from the finding of the previous bake-off study comparing different distributed particle advection strategies [BPYC20]. HyLiPoD is short for Hybrid Lifeline and Parallelize over Data, and the algorithm aims to choose the best strategy between the Lifeline algorithm [BPNC19] and parallelize over data for distributed particle advection given a certain workload.

Finally, Xu et al. [XGS*22] presented an approach that achieved load balancing using reinforcement learning (RL). They introduced three novelties in their work for particle advection: a work donation algorithm, a workload estimation algorithm, and a communication cost estimation model. The RL-agent is responsible for work donation based on the predicted workloads and the communication costs for the current state of the system. Based on the costs the RL-agent either decides to communicate particles or the data blocks to wherever they might be required for a more evenly balanced load. The improvements resultant of these contributions over a baseline algorithm varied between 2X to 36X for the different datasets considered.

4.2.2. Summary

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.

Table 4 provides a simple lookup for a parallelization strategy

Table 5: *Summary of studies considering optimizations for large scale distributed particle advection. The numbers in parenthesis in the Architecture column represent the total number of cores available on the execution platform. The keys to application: SL - streamlines, PL - pathlines, SS - stream surface, S-D - source - destination, STRS - streak surface. The keys to seeding strategy: U - uniform, RD - random distrubution, D - dense, LN - seed line, RK - rakes*

Algo.	Arch.	Procs.	Data size	Time steps	Seed count	App.	Seeding Strategy	Intent / Evaluation
[YWM07]	Intel Xeon (8x4)	32	644M	-	1M	SL, PL	-	hierarchical representation, strong scaling
	AMD Optron (2048x2)	256	644M	100	1M		-	
[CF08]	Intel Xeon (48x2)	32	162M	-	700	SL	-	data partitioning, strong scaling
[PCG*09]	Cray XT5 (ORNL) (149K)	512	512M	-	4k, 22K	SL	U	data loading, data partitioning, weak scaling
		512	512M	-	10K		U	
		512	512M	-	20K		U	
[PRN*11]	PowerPC-450 (40960x4)	16k	8B	-	128k	SL, PL	RD	domain decomposition, dynamic repartitioning, strong and weak scaling
		32K	1.2B	32	16k		RD	
[CCC*11]	Intel Xeon Dash (SDSC)	-	512M	-	2.5K, 10K	SL	D, U	Effects of storage hierarchy
		-	512M	-	2.5K, 10K			
		-	512M	-	2.5K, 10K			
[CGC*11]	Cray XT4 (NERSC) 9572x4	128	512M	-	2.5K, 10K	SL	D, U	MPI-hybrid parallelism
		128	512M	-	2.5K, 10K			
		128	512M	-	1.5K, 6K			
[NLS11]	PowerPC-450 (1024x4)	4K	2B	-	256K	SL	RD	workload aware domain decomposition, strong and weak scaling
		4K	1.2B	-	128K		RD	
[NLL*12]	PowerPC-450 (40960x4)	1k	8M	29	186M	FTLE	U	pipelined temporal advection, caching, strong and weak scaling
		1K	25M	48	65.2M		U	
		16k	345M	36	288M		U	
		16K	43.5M	50	62M		U	
[CCG*12]	Cray XT4 (NERSC) (9572x4)	128	512M	-	128	SS	RK	comparison of parallelization algorithms for stream surfaces
		128	512M	-	361		RK	
		128	512M	-	128		RK	
[MCHG13]	AMD Magny-Cours (6384x24)	1K	32M	735	1M	SL, PL	U	work requesting load balancing, strong scaling
[CBP*14]	Nvidia Kepler (1 GPU / Proc)	8	1B	-	8M	S-D	U	distriburted particle advection over different hardware architectures comparison, strong scaling
	Intel Xeon	192	1B	-	8M			
[GZL*14]	Intel Xeon (8x8)	64	755M	100	-	STRS	LN	sparse data management, strong scaling
		64	3.75M	24	200	PL,	U	
	Intel Xeon (700x12)	512	25M	48	-	FTLE	U	
[LSP14]	PowerPC A2 (2048x16)	1K	25M	-	32K	SS	RK	caching, performance, strong scaling
		4K	80M	-	32K		RK	
		8K	500M	-	32K		RK	
		8K	2B	-	64K		RK	

Algo.	Arch.	Procs.	Data size	Time steps	Seed count	App.	Seeding Strategy	Intent / Evaluation
[ZGY16]	Intel Xeon (8x8)	64 64	3.75M 25M	24 48	6250 4096	PL	U -	data prefetching, strong scaling
[ZGH*17]	PowerPC A2 (2048x16)	8K 8K 8K	1B 3.8M 25M	- 24 48	128M 8M 24M	SL, S-D, FTLE	- - U	domain decomposition, using K-d trees, strong and weak scaling
[HZY18]	Intel Xeon (48) + Nvidia M6000 (8)	48 48 48	25M 3.8M 83M	48 24 28	500K 87.5K 800K	PL PL PL	- - -	"LSTM" for block prediction and prefetching
[BPC19]	Intel Xeon (2388x32)	512	67M	-	1M	S-D	D, U	in situ parallelization over particles
[BPYC20]	Intel Xeon (2388x32)	1K (8K cores)	34B	- -	343M	S-D	D, U	comparison of parallelization algorithms
[BPC21]	Intel Xeon (2388x32)	1K (8K cores)	34B	- -	343M	S-D	D, U	novel hybrid parallelization algorithm
[SCP21]	Nvidia V100 (64)	64	-	-	-	S-D	-	Neural Networks and Random Forests for execution parameters
[MPG*21]	Intel Xeon (4392 x 64)	8K	134M	-	536M	S-D	U	novel algorithm for asynchronous communication and termination detection
[XGS*22]	Intel Xeon (664x32) Intel Xeon (4392x64)	1K 1K 1K 16K	134M 8.7M 25M 69B	- 36 48 -	2M 2.7M 25M 134.2M	SL PL FTLE SL	U D U U	"Reinforcement Learning" for efficient load balancing, work estimation, and communication

Table 6: *Number of particles used per one thousand cells of data for different applications from works described in Table 5.*

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

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 is large. However, in the presence of flow features like critical points and vortices, parallelize over data can suffer from load imbalance. While several methods have been proposed for data repartitioning for load-balanced computation, these works incur the cost of pre-processing and redistributing data. ***Parallelize over particles*** is best suited when the number of particles is large. It can suffer from load imbalance due to inconsistencies in the computational work for different particles. Some works aim to address the problem of load imbalance 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 4 shows 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) [BPNC19], and supervisor-worker (a hybrid parallel algorithm). The figure presents a weak scaling of these algorithms. The top row plots show the throughput of each algorithm in terms of the number of particle advection steps completed by each MPI rank per second. For exam-

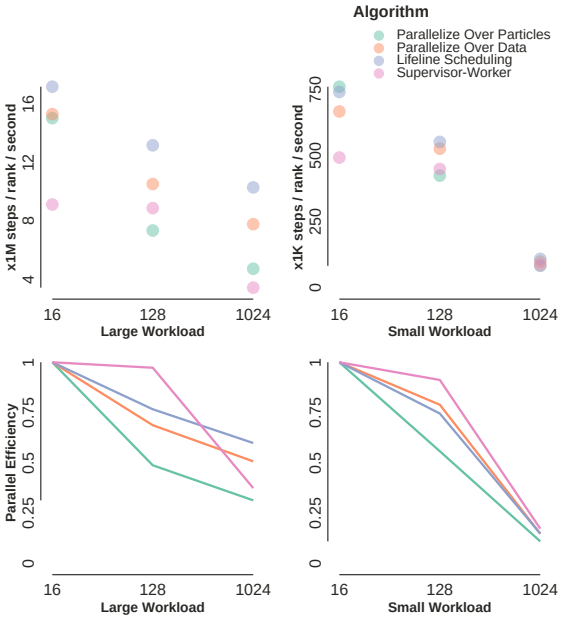


Figure 4: *Weak scaling plots comparing the performance of distributed-memory particle advection algorithms. The plots are presented in a 2x2 arrangement, with the columns organized by workload and the rows organized by efficiency measure. For each of the four plots, the X-Axis is the number of MPI ranks, the Y-Axis is an efficiency measure, and the algorithms are represented as colored dots or lines. With respect to workload, the left column represents a "large" workload (1 particle per 100 cells, advancing for 10K steps) while the right column represents a "small" workload (1 particle per 10K cells, advancing for 1K steps). With respect to efficiency measure, the top row shows the number of steps per rank per second (measured in millions of steps), while the bottom row shows parallel efficiency (the dropoff in steps per second relative to the lowest number of MPI ranks). These plots show that when going from 16 MPI ranks to 1024 MPI ranks, the parallel efficiency drops by a factor of approximately two for large workloads and by a factor of approximately eight for small workloads. Further, some algorithms achieve better performance than others, in particular the Lifeline Scheduling Method. The data for these plots are derived from a study by Binyahib et al. [BPYC20].*

ple, for a large workload and 1024 MPI ranks, the LSM algorithm (shown in purple) performed about 10 million advection steps per second on each rank. 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 are increased. The drop is more significant in smaller workloads than in larger workloads. The only study which compared the scaling behaviors of the most widely used parallelization algorithm used 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 5 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 workload in this survey is by Chen et al. [CF08], which used only 32 processes to produce 700 streamlines. The work with the largest number of processes was by Nouanesengsy et al. [NLL*12], which used 16 thousand processes for FTLE calculation. However, the work with the most workload was by Binyahib et al. [BPYC20], which used 343 million particles for advection in data with 34 billion cells.

Table 6 summarizes the number of particles used in proportion to the size of the data used in the works included in Table 5. 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.

4.3. Hybrid Parallelism for Particle Advection

Hybrid parallelism refers to a combination of using shared- and distributed-memory parallel techniques. For these works, the distributed-memory elements managed dividing work among nodes, and the shared-memory parallelism approach was providing a "pool" of cores that could advect particles quickly. Camp et al. [CGC*11] provided algorithms using multi-core processors for 1) parallelization over particles, and 2) parallelize over data blocks. For parallelization over particles, a total of $2N$ threads are allocated, N worker threads, and N I/O threads. Each worker thread is responsible for performing particle advection and each I/O thread is responsible for managing the cache of data blocks and 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 other cases, 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 be-

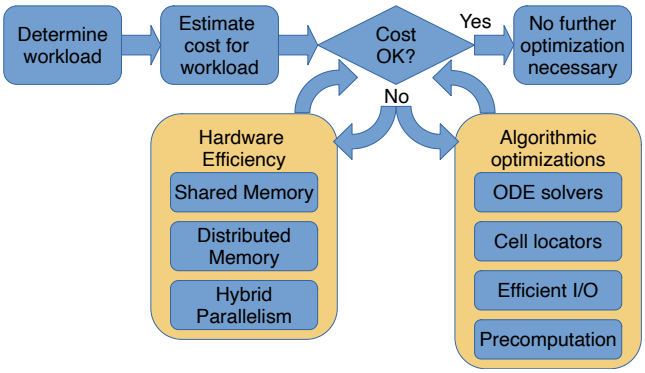


Figure 5: *A proposed cost model-based workflow for deciding on which optimizations to incorporate for a flow visualization algorithm.*

tween 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 for small workloads with short execution times. 3) With higher particle densities (50^3 or more) GPUs can be saturated and result in performance improvements proportional to their FLOP rates, faster GPUs can provide better speedups.

Jiang et al. [JEHG14] studied shared memory multi-threaded generation of streamlines with a locally attached NVRAM. Their particular area of interest was in understanding data movement strategies that will keep the threads busy performing particle advection. They used two data management strategies. The first used explicit I/O to access data. The second was a kernel-managed implicit I/O method that used memory-mapping to provide access to data. Their study indicated that thread over-subscription of streamline tasks is an effective method for hiding I/O latency, which is a bottleneck for particle advection.

Finally, Liao et al. [LMKK18] presented a hybrid-parallel implementation of 3D Line Integral Convolutions (LIC). Their technique involved parallelizing ray casting operations using multi-threading with OpenMP. During the process, LIC values are sampled serially as voxel intensity for each ray, and each sample involves seeding a particle to trace an integral field line and convolve noise values. They reported better strong scaling for both shared- and distributed-memory parallelism for thier LIC implementation compared to streamline generation.

5. Conclusion and Future Work

This state-of-the-report has considered optimizing particle advection performance, surveying existing approaches for algorithmic optimizations and parallelism. Looking ahead to future work, we feel this STAR has illuminated three types of gaps in the area of particle advection performance.

The first type of gap involves the lack of holistic studies to inform behavior across diverse workloads. Adaptive step sizing, since its focus is more on accuracy than performance, can lead to highly varying speedups. Understanding when speedups occur and their magnitude would be very helpful for practitioners when deciding whether to include this approach. Similarly, the expected speedup for a GPU is highly varied based on workload and GPU architecture. While this survey was able to synthesize results from a recent study [PYK*18], significantly more detail would be useful.

The second type of gap covers possible optimizations that have not yet been pursued. All of the hardware efficiency works in this survey involved parallelism, yet there are still additional hardware optimizations available. In the ray tracing community — similar to particle advection in that rays move through a volume in a data-dependent manner — packet tracing, where rays on similar trajectories are traced together, has led to significant speedups. Further, there can be significant improvement from complex schemes. For example, Benthin et al. [BWW*12] employed a hybrid approach that generates and traces rays in packets and then automatically switches to tracing rays individually when they diverge. This hybrid algorithm outperforms conventional packet tracers by up to 2X. A related point that needs further exploration is the impact of synchronization and locking of shared memory on GPUs. Synchronization allows for better coordination, for example, when tracing coherent rays through a volume. Understanding the impact on particle advection is an area that needs further work. Finally, there are additional types of optimizations. Taking another example from ray tracing, Morrical et al. [MWUP20] presented a method that improved the performance of direct unstructured mesh point location [SC20] by using the Nvidia RTX GPU. Their approach re-implemented the point location problem as a ray tracing problem, which enabled tracing the points using the hardware. Their results showed equal or better performance compared to state-of-the-art solutions and could provide inspiration for improved cell locators on GPUs for particle advection.

The third type of gap is in cost modeling. One benefit for cost modeling would be using prediction to adapt workloads to fit available runtime. A second (perhaps more powerful) benefit would be to enable a workflow for decision-making. This workflow is shown via a flowchart in Figure 5, and would operate in three steps. In the first step, the desired workload would be analyzed to see how many operations need to be performed. In the second step, the analysis from the first step would be used to estimate the execution time costs to execute the algorithm. In the third step, the estimated costs from the second step would be 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 candidate optimizations should be considered and the workflow should be repeated with candidate optimizations until the desired runtime is predicted. Realizing such a cost model, however, could be quite challenging, including I/O, precomputation, parallel speedups, and more.

References

[ACG*14] AGRANOVSKY A., CAMP D., GARTH C., BETHEL E. W., JOY K. I., CHILDS H.: Improved Post Hoc Flow Analysis via Lagrangian Representations. In *2014 IEEE 4th Symposium on Large Data Analysis and Visualization (LDAV)* (2014), IEEE, pp. 67–75. 6

[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. 12

[AT10] ANDRYSCO N., TRICOCHE X.: Matrix Trees. *Comput. Graph. Forum 29*, 3 (2010), 963–972. 5

[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). 7

[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. 7

[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. 7, 9

[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. 13, 15

[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. 7, 9

[BL99] BLUMOFE R. D., LEISERSON C. E.: Scheduling Multithreaded Computations by Work Stealing. *Journal of the ACM (JACM) 46*, 5 (1999), 720–748. 12

[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. 13, 15

[BPC21] BINYAHIB R., PUGMIRE D., CHILDS H.: HyLiPoD: Parallel Particle Advection Via a Hybrid of Lifeline Scheduling and Parallelization-Over-Data. 13, 15

[BPNC19] BINYAHIB R., PUGMIRE D., NORRIS B., CHILDS H.: A Lifeline-Based Approach for Work Requesting and Parallel Particle Advection. In *2019 IEEE 9th Symposium on Large Data Analysis and Visualization (LDAV)* (2019), IEEE, pp. 52–61. 13, 15

[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. 13, 15, 16

[BRKE*11] BUSSLER M., RICK T., KELLE-EMDEN A., HENTSCHEL B., KUHLEN T.: Interactive Particle Tracing in Time-Varying Tetrahedral Grids. In *Eurographics Symposium on Parallel Graphics and Visualization* (2011), Kuhlen T., Pajarola R., Zhou K., (Eds.), The Eurographics Association. 5, 8, 9

[BSGC17] BLEILE R., SUGIYAMA L., GARTH C., CHILDS H.: Accelerating Advection via Approximate Block Exterior Flow Maps. *Electronic Imaging 2017*, 1 (2017), 140–148. 6

[BSK*07] BUERGER K., SCHNEIDER J., KONDRATIEVA P., KRUEGER J., WESTERMANN R.: Interactive Visual Exploration of Unsteady 3D Flows. In *Eurographics/ IEEE-VGTC Symposium on Visualization* (2007), The Eurographics Association. 7, 9

[BT13] BARAKAT S. S., TRICOCHE X.: Adaptive refinement of the flow map using sparse samples. *IEEE Transactions on Visualization and Computer Graphics 19*, 12 (2013), 2753–2762. 6

[BWW*12] BENTHIN C., WALD I., WOOP S., ERNST M., MARK W. R.: Combining Single and Packet-Ray Tracing for Arbitrary Ray Distributions on the Intel MIC Architecture. *IEEE Transactions on Visualization and Computer Graphics 18*, 9 (2012), 1438–1448. 17

[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, 16

[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. 12, 14

[CCG*12] CAMP D., CHILDS H., GARTH C., PUGMIRE D., JOY K. I.: Parallel Stream Surface Computation for Large Data Dets. In *IEEE Symposium on Large Data Analysis and Visualization (LDAV)* (2012), IEEE, pp. 39–47. 14

[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. 11, 14, 16

[CGC*11] 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. 10, 11, 12, 14, 16

[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. 16

[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. 5, 6, 7, 12

[COJ15] CHANDLER J., OBERMAIER H., JOY K. I.: Interpolation-Based Pathline Tracing in Particle-Based Flow Visualization. *IEEE Transactions on Visualization and Computer Graphics 21*, 1 (2015), 68–80. 6

[com08] *Computational Geometry*. Springer Berlin, Heidelberg, 2008. 4

[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. 12

[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. 5, 6, 7, 12

[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. 5, 6, 7, 12

[DG08] DEAN J., GHEMAWAT S.: MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM 51*, 1 (2008), 107–113. 13

[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. 12

[DLYC15] DING Z., LIU Z., YU Y., CHEN W.: Parallel Unsteady Flow Line Integral Convolution for High-Performance Dense Visualization. In *2015 IEEE Pacific Visualization Symposium (PacificVis)* (2015), pp. 25–30. doi:10.1109/PACIFICVIS.2015.7156352. 8

[Ern21] ERN, ALEXANDRE AND GUERMOND, JEAN-LUC: *Finite Elements I*. Springer Cham, 2021. 4

[Far02] FARIN, GERALD: *Curves and Surfaces for Computer-Aided Geometric Design*. Morgan-Kaufmann, 2002. 4

[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. 3

[GGW22] GHAFFARI B., GATTI D., WESTERMANN R.: Spatio-Temporal Visual Analysis of Turbulent Superstructures in Unsteady Flow. *IEEE Transactions on Visualization and Computer Graphics* (2022), 1–14. 8

[GHP*16] GUO H., HE W., PETERKA T., SHEN H.-W., COLLIS S. M., HELMUS J. J.: Finite-Time Lyapunov Exponents and Lagrangian Coherent Structures in Uncertain Unsteady Flows. *IEEE Transactions on Visualization and Computer Graphics 22*, 6 (2016), 1672–1682. 8

[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. 13

[GHS*19] GUO H., HE W., SEO S., SHEN H.-W., CONSTANTINESCU E. M., LIU C., PETERKA T.: Extreme-Scale Stochastic Particle Tracing for Uncertain Unsteady Flow Visualization and Analysis. *IEEE Transactions on Visualization and Computer Graphics 25*, 9 (2019), 2710–2724. 8

[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. 4, 5, 8, 9

[GKT16] GÜNTHER T., KUHN A., THEISEL H.: MCFTLE: Monte Carlo Rendering of Finite-Time Lyapunov Exponent Fields. *Comput. Graph. Forum 35*, 3 (jun 2016), 381–390. 8

[GLS99] GROPP W. D., LUSK E., SKJELLUM A.: *Using MPI: portable parallel programming with the message-passing interface*, vol. 1. MIT press, 1999. 10

[GSBW12] GEVECI B., SCHROEDER W., BROWN A., WILSON G.: VTK. *The Architecture of Open Source Applications 1* (2012), 387–402. 3

[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. 13

[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. 13

[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. 12, 14

[HGK*15] HENTSCHEL B., GÖBBERT J. H., KLEMM M., SPRINGER P., SCHNORR A., KUHLEN T. W.: Packet-Oriented Streamline Tracing on Modern SIMD Architectures. In *Eurographics Symposium on Parallel Graphics and Visualization* (2015), Dachsbacher C., Navrátil P., (Eds.), The Eurographics Association. 8

[Hil91] HILBERT D.: Über die stetige abbildung einer linie aufein flächenstück. *Mathematische Annalen 38* (1891), 459–460. 5

[HMCA15] HANWELL M. D., MARTIN K. M., CHAUDHARY A., AVILA L. S.: The Visualization Toolkit (VTK): Rewriting the rendering code for modern graphics cards. *SoftwareX 1* (2015), 9–12. 3

[HNW00] HAIRER E., NØRSETT S., WANNER G.: *Solving Ordinary Differential Equations I Nonstiff problems*, second ed. Springer, Berlin, 2000. 3

[HSJ22] HAN M., SANE S., JOHNSON C. R.: Exploratory lagrangian-based particle tracing using deep learning. *Journal of Flow Visualization and Image Processing 29*, 3 (2022), 73–96. 6

[HSW10] HLAWATSCH M., SADLO F., WEISKOPF D.: Hierarchical Line Integration. *IEEE Transactions on Visualization and Computer Graphics 17*, 8 (2010), 1148–1163. 6

[HZY18] HONG F., ZHANG J., YUAN X.: Access Pattern Learning with Long Short-Term Memory for Parallel Particle Tracing. In *2018 IEEE Pacific Visualization Symposium (PacificVis)* (2018), pp. 76–85. 5, 15

[JEHG14] JIANG M., ESSEN B. V., HARRISON C., GOKHALE M. B.: Multi-Threaded Streamline Tracing for Data-Intensive Architectures. In *4th IEEE Symposium on Large Data Analysis and Visualization, LDAV 2014, Paris, France, November 9-10, 2014* (2014), Childs H., Pajarola R., Vishwanath V., (Eds.), IEEE Computer Society, pp. 11–18. 17

[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. 7, 9

[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. 5

[Kno06] KNOLL A.: A Survey of Octree Volume Rendering Methods. In *Visualization of Large and Unstructured Data Sets* (2006), Hagen H., Kerren A., Dannenmann P., (Eds.). 4

[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. 3

[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* (New York, NY, USA, 2011), SC '11, ACM, pp. 10:1–10:11. 13

[LA90] LÖHNER R., AMBROSIANO J.: A Vectorized Particle Tracer for Unstructured Grids. *Journal of Computational Physics 91*, 1 (1990), 22–31. 4, 5, 6, 7

[Lan95] LANE D. A.: *Parallelizing A Particle Tracer for Flow Visualization*. Tech. rep., Society for Industrial and Applied Mathematics, Philadelphia, PA (United States), 1995. 8

[Lex15] LEXI C.: Application-Specific: Polar, Far-Field, and Particle Tracing Plots, 2015. URL: https://www.comsol.com/blogs/application-specific-polar-far-field-and-partic\le-tracing-plots. 3

[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. 13

[LHD*04] LARAMEE R. S., HAUSER H., DOLEISCH H., VROLIJK B., POST F. H., WEISKOPF D.: The State of the Art in Flow Visualization: Dense and Texture-Based Techniques. In *Computer Graphics Forum* (2004), vol. 23, Wiley Online Library, pp. 203–221. 1

[LHZP07] LARAMEE R. S., HAUSER H., ZHAO L., POST F. H.: Topology-Based Flow Visualization, The State of the Art. In *Topology-based methods in visualization*. Springer, 2007, pp. 1–19. 1

[LM05] LIU Z., MOORHEAD R.: Accelerated Unsteady Flow Line Integral Convolution. *IEEE Transactions on Visualization and Computer Graphics 11*, 2 (2005), 113–125. doi:10.1109/TVCG.2005.21. 8

[LMKK18] LIAO Y., MATSUI H., KREYLOS O., KELLOGG L. H.: A Flow Visualization Using Parallel 3D Line Integral Convolution for Large Scale Unstructured Grid Data. In *2018 IEEE 8th Symposium on Large Data Analysis and Visualization (LDAV)* (2018), pp. 106–107. doi:10.1109/LDAV.2018.8739209. 17

[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. 12, 14

[LST03] LANGBEIN M., SCHEUERMANN G., TRICOCHE X.: An Efficient Point Location Method for Visualization in Large Unstructured Grids. In *Proceedings of Vision, Modeling, Visualization* (2003). 4, 5

[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. 13, 14

[MLP*10] McLoughlin T., Laramee R. S., Peikert R., Post F. H., Chen M.: Over Two Decades of Integration-Based, Geometric Flow Visualization. In *Computer Graphics Forum* (2010), vol. 29, Wiley Online Library, pp. 1807–1829. 2

[MPG*21] Morozov D., Peterka T., Guo H., Raj M., Xu J., Shen H.-W.: IExchange: Asynchronous Communication and Termination Detection for Iterative Algorithms. In *2021 IEEE 11th Symposium on Large Data Analysis and Visualization (LDAV)* (2021), pp. 12–21. 12, 15

[MSU*16] Moreland K., Sewell C., Usher W., Lo L.-t., Meredith J., Pugmire D., Kress J., Schroots H., Ma K.-L., Childs H., Larsen M., Chen C.-M., Maynard R., Geveci B.: VTK-m: Accelerating the Visualization Toolkit for Massively Threaded Architectures. *IEEE Computer Graphics and Applications 36*, 3 (2016), 48–58. doi:10.1109/MCG.2016.48. 8

[MWUP20] Morrical N., Wald I., Usher W., Pascucci V.: Accelerating Unstructured Mesh Point Location with RT Cores. *IEEE Transactions on Visualization and Computer Graphics* (2020). 17

[Nic07] Nickolls J.: GPU Parallel Computing Architecture and CUDA Programming Model. In *2007 IEEE Hot Chips 19 Symposium (HCS)* (2007), IEEE, pp. 1–12. 7

[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, pp. 1–11. 11, 14, 16

[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. 5, 11, 14

[OPF*12] Özgökmen T. M., Poje A. C., Fischer P. F., Childs H., Krishnan H., Garth C., Haza A. C., Ryan E.: On Multi-Scale Dispersion Under the Influence of Surface Mixed Layer Instabilities. *Ocean Modelling 56* (Oct. 2012), 16–30. 1

[Pan19] Panda D. K.: Overview of the mvapich project: Latest status and future roadmap. In *MVAPICH2 User Group (MUG) Meeting* (2019). 10

[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. 13, 14

[PD81] Prince P. J., Dormand J. R.: High Order Embedded Runge-Kutta Formulae. *Journal of Computational and Applied Mathematics 7*, 1 (1981). 3

[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. 3

[PPF*11] Pobitzer A., Peikert R., Fuchs R., Schindler B., Kuhn A., Theisel H., Matković K., Hauser H.: The State of the Art in Topology-Based Visualization of Unsteady Flow. In *Computer Graphics Forum* (2011), vol. 30, Wiley Online Library, pp. 1789–1811. 1

[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. 11, 13, 14

[PSCB21] Panda D. K., Subramoni H., Chu C.-H., Bayatpour M.: The MVAPICH project: Transforming research into high-performance MPI library for HPC community. *Journal of Computational Science 52* (2021), 101208. Case Studies in Translational Computer Science. 10

[PVH*03] Post F. H., Vrolijk B., Hauser H., Laramee R. S., Doleisch H.: The State of the Art in Flow Visualisation: Feature Extraction and Tracking. In *Computer Graphics Forum* (2003), vol. 22, Wiley Online Library, pp. 775–792. 1

[PWK21] Preus D., Weinkauf T., Kruger J.: A Discrete Probabilistic Approach to Dense Flow Visualization. *IEEE Transactions on Visualization & Computer Graphics 27*, 12 (dec 2021), 4347–4358. 8

[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. 8, 9, 10, 17

[RD20] Rapp T., Dachsbacher C.: Uncertain Transport in Unsteady Flows. In *31st IEEE Visualization Conference, IEEE VIS 2020 - Short Papers, Virtual Event, USA, October 25-30, 2020* (2020), IEEE, pp. 16–20. 8

[RGG20] Rojo I. B., Gross M. H., Günther T.: Accelerated Monte Carlo Rendering of Finite-Time Lyapunov Exponents. *IEEE Transactions on Visualization and Computer Graphics 26* (2020), 708–718. 8

[RPD19] Rapp T., Peters C., Dachsbacher C.: Void-and-Cluster Sampling of Large Scattered Data and Trajectories. *IEEE transactions on visualization and computer graphics 26*, 1 (2019), 780–789. 6

[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. 12

[SBGC20] Sane S., Bujack R., Garth C., Childs H.: A Survey of Seed Placement and Streamline Selection Techniques. In *Computer Graphics Forum* (2020), vol. 39, Wiley Online Library, pp. 785–809. 2

[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. 5, 8, 9

[SC20] Sawhney R., Crane K.: Monte Carlo Geometry Processing: A Grid-Free Approach to PDE-Based Methods on Volumetric Domains. *ACM Trans. Graph. 39*, 4 (July 2020). 17

[SCB19] Sane S., Childs H., Bujack R.: An Interpolation Scheme for VDVP Lagrangian Basis Flows. In *EGPGV@ EuroVis* (2019), pp. 109–119. 6

[SCP21] Schwartz S., Childs H., Pugmire D.: Machine Learning-Based Autotuning for Parallel Particle Advection. In *Eurographics Symposium on Parallel Graphics and Visualization (EGPGV)* (Zurich, Switzerland, June 2021), pp. 7–17. 12, 15

[SCT*10] Sanderson A. R., Chen G., Tricoche X., Pugmire D., Kruger S., Breslau J. A.: Analysis of Recurrent Patterns in Toroidal Magnetic Fields. *IEEE Trans. Vis. Comput. Graph. 16*, 6 (2010), 1431–1440. 4

[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). 10

[SML97] Schroeder W., Martin K., Lorensen B.: *The Visualization Toolkit: An Object-Oriented Approach to 3D Graphics*. Prentice Hall, 1997. 4

[SP16] Sisneros R., Pugmire D.: Tuned to Terrible: A Study of Parallel Particle Advection State of the Practice. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)* (2016), pp. 1058–1067. doi:10.1109/IPDPSW.2016.173. 12

[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. 12

[SYB*21] Sane S., Yenpure A., Bujack R., Larsen M., Moreland K., Garth C., Johnson C. R., Childs H.: Scalable In Situ Computation of Lagrangian Representations via Local Flow Maps. In *Eurographics Symposium on Parallel Graphics and Visualization* (2021), Larsen M., Sadlo F., (Eds.), The Eurographics Association. 6

[top20] Top 500, 2020. URL: https://www.top500.org/lists/top500/2020/11. 7

[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. 5, 6, 7

[WvG92] WILHELMS J., VAN GELDER A.: Octrees for Faster Isosurface Generation. *ACM Transactions of Graphics 11*, 3 (1992), 201–227. 4

[XGS*22] XU J., GUO H., SHEN H.-W., RAJ M., WURSTER S. W., PETERKA T.: Reinforcement Learning for Load-Balanced Parallel Particle Tracing. *IEEE Transactions on Visualization and Computer Graphics* (2022). 13, 15

[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. 11, 14

[ZGH*17] ZHANG J., GUO H., HONG F., YUAN X., PETERKA T.: Dynamic Load Balancing Based on Constrained K-D Tree Decomposition for Parallel Particle Tracing. *IEEE transactions on visualization and computer graphics 24*, 1 (2017), 954–963. 11, 12, 15

[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. 12, 15

[ZGYP18] ZHANG J., GUO H., YUAN X., PETERKA T.: Dynamic Data Repartitioning for Load-Balanced Parallel Particle Tracing. *2018 IEEE Pacific Visualization Symposium (PacificVis)* (2018), 86–95. 11

[ZY18] ZHANG J., YUAN X.: A Survey of Parallel Particle Tracing Algorithms in Flow Visualization. *Journal of Visualization 21*, 3 (2018), 351–368. 1

[ZZ94] ZHANG R., ZHANG C.-T.: Z Curves, an Intutive Tool for Visualizing and Analyzing the DNA Sequences. *Journal of Biomolecular Structure and Dynamics 11*, 4 (1994), 767–782. 5

[ZZO03] ZHANG C.-T., ZHANG R., OU H.-Y.: The Z Curve Database: A Graphic Representation of Genome Sequences. *Bioinformatics 19*, 5 (2003), 593–599. 5