GENERAL PURPOSE FLOW VISUALIZATION AT THE EXASCALE

by

ABHISHEK DILIP YENPURE

A DISSERTATION

Presented to the Department of Computer Science
and the Division of Graduate Studies of the University of Oregon
in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy

December 2022

DISSERTATION APPROVAL PAGE

Student: Abhishek Dilip Yenpure

Title: General Purpose Flow Visualization at the Exascale

This dissertation has been accepted and approved in partial fulfillment of the requirements for the Doctor of Philosophy degree in the Department of Computer Science by:

Hank Childs	Chair
Boyana Norris	Core Member
Jeewhan Choi	Core Member
Ellen Eischen	Institutional Representative

and

Krista Chronister	Vice Provost of Graduate Studies

Original approval signatures are on file with the University of Oregon Division of Graduate Studies.

Degree awarded December 2022

© 2022 Abhishek Dilip Yenpure
This work, including text and images of this document but not including supplemental files (for example, not including software code and data), is licensed under a Creative Commons
Attribution 4.0 International License.

DISSERTATION ABSTRACT

Abhishek Dilip Yenpure

Doctor of Philosophy

Department of Computer Science

December 2022

Title: General Purpose Flow Visualization at the Exascale

Exascale computing, i.e., supercomputers that can perform 10^{18} math operations per second, provide significant opportunity for improving the computational sciences. That said, these machines can be difficult to use efficiently, due to their massive parallelism, due to the use of accelerators, and due to the diversity of accelerators used. All areas of the computational science stack need to be reconsidered to address these problems. With this dissertation, we consider flow visualization, which is critical for analyzing vector field data from simulations. We specifically consider flow visualization techniques that use particle advection, i.e., tracing particle trajectories, which presents performance and implementation challenges. The dissertation makes four primary contributions. First, it synthesizes previous work on particle advection performance and introduces a high-level analytical cost model. Second, it proposes an approach for performance portability across accelerators. Third, it studies expected speedups based on using accelerators, including the importance of factors such as duration, particle count, data set, and others. Finally, it proposes an exascale-capable particle advection system that addresses diversity in many dimensions, including accelerator type, parallelism approach, analysis use case, underlying vector field, and more.

CURRICULUM VITAE

NAME OF AUTHOR: Abhishek Dilip Yenpure

GRADUATE AND UNDERGRADUATE SCHOOLS ATTENDED:

University of Oregon, Eugene, OR, USA
University of Pune, Pune, India

DEGREES AWARDED:

Doctor of Philosophy, Computer Science, 2022, University of Oregon
Master of Science, Computer Science, 2018, University of Oregon
Bachelor of Engineering, Information Technology, 2013, University of Pune

AREAS OF SPECIAL INTEREST:

Flow Visualization
Scientific Visualization
High Performance Computing

PROFESSIONAL EXPERIENCE:

Senior R&D Engineer, Kitware, April 2022-Present
Graduate Research Fellow, University of Oregon, Jan 2017 - March 2022
Graduate Teaching Fellow, University of Oregon, Fall 2016, Spring 2021
Summer Intern, National Center for Atmospheric Research, Summer 2020
Summer Intern, Sandia National Laboratories, Summer 2018, 2019
Summer Intern, Oak Ridge National Laboratories, Summer 2017
Senior Software Engineer, eQ Technologic, March 2016 - August 2016
Software Engineer, eQ Technologic, June 2013 - March 2016

GRANTS, AWARDS AND HONORS:

Best Paper Award, "Scalable In Situ Computation of Lagrangian Representations via Local Flow Maps" At the Eurographics Symposium on Parallel Graphics and Visualization (EGPGV), Zurich, Switzerland, June 2021

J. Donald Hubbard Family Scholarship, University of Oregon, 2020

Sushil Jajodia Endowed Scholarship, University of Oregon, 2017, 2019

Jhamandas Watumull Endowed Scholarship, University of Oregon, 2016

PUBLICATIONS:

D Pugmire, A Yenpure, M Kim, J Kress, R Maynard, H Childs, B Hentschel, “Performance-Portable Particle Advection with VTK-m,” *Eurographics Symposium on Parallel Graphics and Visualization*, 2018

A Yenpure, H Childs, K Moreland, “Efficient Point Merging Using Data Parallel Techniques,” *Eurographics Symposium on Parallel Graphics and Visualization*, 2019

R Binyahib, D Pugmire, A Yenpure, H Childs, “Parallel Particle Advection Bake-Off for Scientific Visualization Workloads,” *IEEE International Conference on Cluster Computing (CLUSTER)*, 2020

S Sane, A Yenpure, R Bujack, M Larsen, K Moreland, C Garth, C Johnson, H Childs, “Scalable In Situ Computation of Lagrangian Representations via Local Flow Maps,” Eurographics Symposium on Parallel Graphics and Visualization, 2021

K Moreland, R Maynard, D Pugmire, A Yenpure, A Vacanti, M Larsen, H Childs, “Minimizing development costs for efficient many-core visualization using MCD3,” Parallel Computing, 2021

A Yenpure, S Sane, R Binyahib, D Pugmire, C Garth, H Childs, “A Guide to Particle Advection Performance.” (In submission)

A Yenpure, D Pugmire, H Childs, “Steady State Particle Advection on GPUs and CPUs.” (In preparation)

A Yenpure, D Pugmire, C Garth, H Childs, “PICS: A Parallel Integral Curve System.” (In preparation)

ACKNOWLEDGEMENTS

I want to thank my advisor Prof. Hank Childs for his constant support, patience, and understanding during my tenure as a Ph.D. student. His guidance and advice have been instrumental in helping me to complete my degree requirements and have allowed me to build relationships that will help me throughout my career.

I'd also like to thank Dr. David Pugmire, Dr. Kenneth Moreland, and Dr. John Clyne for mentoring me during my internships during my graduate studies. These internships have been invaluable and helped me grow personally and professionally.

Finally, I'd like to thank all the previous and current members of the CDUX research group for all the help I received from them over the years.

This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration.

To Mom and Dad. Their love, encouragement, and support have given me the strength and courage to face any challenge with a smile.

TABLE OF CONTENTS

Chapter Page

I. INTRODUCTION 1

1.1. Introduction 1

1.1.1. Exascale Computing 2

1.1.2. Large Scale Visualization 3

1.1.3. Flow Visualization 5

1.1.4. Thesis Question 7

1.2. Performant Particle Advection System 9

1.2.1. Studies for Intrinsic Factors 9

1.2.1.1. Algorithmic Optimizations and Cost Modeling . . . 10

1.2.2. Studies for Extrinsic Factors 11

1.2.2.1. Performance Portability 12

1.2.2.2. Particle Advection Speedups from GPU and CPU Parallelism 12

1.3. General Purpose Flow Visualization System for Exascale 14

II. BACKGROUND 16

2.1. Introduction 17

2.2. Particle Advection Background 19

2.3. Cost Model for Particle Advection Performance 23

2.4. Algorithmic Optimizations 25

2.4.1. ODE Solvers 26

2.4.2. Cell Locators 27

2.4.3. I/O Efficiency 30

2.4.4. Precomputation 31

Chapter Page

2.4.5. Summary 33

2.5. Using Hardware Efficiently 35

2.5.1. Shared Memory Parallelism for Particle Advection 35

2.5.1.1. Shared memory GPUs 37

2.5.1.2. Shared Memory Parallelism on CPUs 38

2.5.1.3. Summary 39

2.5.2. Distributed Memory Parallelism for Particle Advection . . . 39

2.5.2.1. Parallelization Methods 42

2.5.2.2. Summary 55

2.5.3. Hybrid Parallelism for Particle Advection 58

2.6. Conclusion and Future Work 59

III. PORTABLE PARTICLE ADVECTION USING VTK-M 63

3.1. Introduction 64

3.2. Related Work 66

3.3. Parallel Particle Advection in VTK-m 70

3.3.1. A Data-Parallel Formulation of Particle Advection 71

3.4. Performance Evaluation 75

3.4.1. Experimental Setup 75

3.4.2. Results 79

3.4.2.1. VTK-m Results 79

3.4.2.2. Comparisons to Other Implementations 82

3.4.3. Discussion 87

3.5. Conclusion & Future Work 89

IV. STEADY STATE PARTICLE ADVECTION ON GPUS AND CPUS 91

4.1. Introduction 91

4.2. Related Work 93
4.3. Experimental Overview 96
4.3.1. Data Sets 97
4.3.2. Workloads 98
4.3.3. Hardware Usage 99
4.4. Results 100
4.4.1. **RQ1**: How Much Speedup Will a GPU Provide Over a Serial CPU Implementation? 101
4.4.1.1. Effect from Number of Advection Steps 101
4.4.1.2. Effect of GPU Architecture 103
4.4.1.3. Effect of Algorithm 104
4.4.1.4. Effect of GPU Transfer Mode 107
4.4.1.5. Synthesis 108
4.4.2. **RQ2**: How Much Speedup Will a Multi-core CPU Provide Over a Serial CPU Implementation? 110
4.4.2.1. Takeaways For CPUs 113
4.4.3. **RQ3**: How Does GPU Performance Compare to Multi-Core CPU Performance? 114
4.4.3.1. Takeaways for GPUs and CPUs 117
4.4.4. **RQ4**: What Are the Trends in Performance by Using Newer Hardware? 117
4.4.5. **RQ5**: How Does Data Set Impact Performance? 120
4.5. Conclusion 127
V. GENERAL PURPOSE FLOW VISUALIZATION FOR THE EXASCALE 131
5.1. Introduction 131
5.2. Related Work 134

5.2.1. Towards Performance Efficiency 134
5.2.2. Towards Developer Efficiency 136
5.3. Implementation 136
5.3.1. Terminology 137
5.3.2. Achieving Developer Efficiency and Generality 138
5.3.2.1. Particle Abstraction 138
5.3.2.2. Evaluator Abstraction 138
5.3.2.3. Field Abstraction 139
5.3.2.4. Analysis Abstraction 141
5.3.2.5. Termination Criteria Abstraction 142
5.3.2.6. Solver Abstraction 142
5.3.3. Organization and Interaction 144
5.3.4. Specifying a VTK-m Flow Filter 144
5.4. Results 145
5.4.1. Streamlines for WarpX 146
5.4.2. Poincaré for XGC 147
5.4.3. FTLE for Cloverleaf 149
5.4.4. Performance Evaluations using Cloverleaf 151
5.5. Conclusion 153
VI. CONCLUSION AND FUTURE WORK 154
6.1. Efficient Approaches for Particle Advection 155
6.1.1. Future Work 157
6.2. Achieving Generality for Flow Visualization 158

APPENDICES

Chapter Page

A. COST MODEL VALIDATION 159

A.1. Evaluating Cost Model Terms in Floating-Point Operations 159

A.2. Notional Usage of the Cost Model 160

A.3. Empirical Validation 162

B. GPU AND CPU PERFORMANCE ON DIFFERENT DATA SETS 164

B.1. CPU Performance on Different Data Sets 164

B.2. GPU Performance on Different Data Sets 165

REFERENCES CITED 167

LIST OF FIGURES

Figure Page

1. Organization of chapters with respect to the dissertation questions . . . 7

2. Abstractions for components of a particle advection based flow visualization algorithm to achieve the “efficiency squared” criteria 13

3. Organization of the components for a particle advection-based flow visualization algorithm, including the order of execution 19

4. Examples of flow visualizations from four representative algorithms 22

5. Weak scaling plots for distributed memory particle advection based on the comparison of four parallelization algorithms 56

6. A flow chart to determine the potential optimizations to be applied to a flow visualization algorithm 60

7. Pseudocode for our implementation of the parallelize-over-seed (POS) strategy 72

8. Pseudocode for our implementation of particle advection using data parallel primitives 74

9. Sample streamlines show typical vector field patterns in the three data sets used in this study. 76

10. Parallel efficiency for the VTK-m implementation running on the CPU of the Rhea cluster. 82

11. Comparison of GPU experiments to serial CPU experiments 102

12. Comparison of speedups between an Ampere GPU and a Kepler GPU for workloads with no data transfer 103

13. Differences between speedups achieved by the particle advection and streamline algorithms 106

Figure Page

14. Slowdown introduced by memory transfer as a function of execution time . . . 107

15. Comparison of multi-core CPU experiments to serial CPU experiments . . . 111

16. Comparison of performance for the 104-core Xeon with the 12-core Xeon . . . 113

17. Comparison of GPU speedups for particle advection over their multi-core CPU counterparts . . . 115

18. Comparison of experiments in combination of the best and worst GPUs and multi-core CPUs their corresponding serial CPU experiments . . . 116

19. Speedups achieved by different generation of GPUs and CPUs for particle advection and streamline algorithms . . . 118

20. Evaluating the proportion of fake steps contributed by different termination criteria when particles are not terminated at all . . . 125

21. The abstractions and their concrete realizations for our general purpose flow visualization system . . . 133

22. The organization of components using the introduced abstractions for a particle advection-based flow visualization filter . . . 143

23. Streamline visualizations for WarpX data . . . 148

24. Poincaré (puncture) plots generated for the XGC app . . . 149

25. Iso-surfaces extracted using 5 different values of the FTLE field generated using CloverLeaf3D's velocity field . . . 150

26. Streamlines generated using the CloverLeaf data using 100 seeds being advected for 10000 steps each. . . . 152

A.1. Comparison of analytical cost and actual execution cost for particle advection . . . 161

B.1. Comparison of CPU performance for different data sets . . . 165

B.2. Comparison of GPU performance for different data sets . . . 166

LIST OF TABLES

Table Page

1. Parameters for seeding strategy, the number of seeds, and the number of steps for four representative flow visualization algorithms 23

2. Summary of studies considering algorithmic optimizations to particle advection 34

3. Summary of studies considering optimizations for shared memory particle advection 40

4. Table from the Binyahib survey on recommended parallelization strategy 52

5. Summary of studies considering optimizations for large scale distributed particle advection 52

6. Number of particles used per one thousand cells of data for different large scale flow visualization algorithms surveyed in Chapter II 55

7. Hardware used in performance portable particle advection study 78

8. Timings for VTK-m implementations for each test in our experimental setup 80

9. Timings for the GPU comparison implementation along with a performance factor for the VTK-m timings 85

10. Timings for the CPU comparison implementation along with a comparison factor to the VTK-m timings 86

11. Timings for the VisIt implementation on two different workloads along with a comparison factor to the VTK-m timings 87

12. The list of CPUs an GPUs that used in Chapter IV 100

13. The 270 configurations used to investigate GPU speedups over a serial CPU 101

Table Page

14. Speedups over serial execution achieved by the particle advection and streamline algorithms for the different workloads while using worst and best GPUs with both transfer modes 109

15. The 150 configurations used to investigate multi-core CPU speedups over serial a CPU 110

16. Speedups over serial execution achieved by the two algorithms for the different workloads while using worst and best CPUs 114

17. The 360 configurations used to investigate GPU speedups over multi-core CPU speedups 114

18. Comparisons between slowest and fastest CPUs and GPUs 117

19. Summary of configurations used to investigate the impact of dataset on particle advection 121

20. Aggregate speedup values for all combinations of hardware architectures and data sets for particle advection 123

21. Relative slowdowns in terms of time per step for different architectures when particles are not terminated at all 124

22. Abstractions and their realizations for the three scientific use cases and the performance study considered to demonstrate the efficacy of the general purpose particle advection system 146

23. Comparison of performance against VisIt for large-scale experiments of streamline visualizations using CloverLeaf data 152

A.1. Analytical cost calculation for particle advection for a hypothetical case 160

CHAPTER I

INTRODUCTION

Most content for this chapter comes from my dissertation proposal. I was the primary author for this chapter, and Hank Childs provided editorial suggestions.

1.1 Introduction

Scientific discoveries are driven by the cycle of hypothesizing followed by experimenting to measure and observe some natural phenomena. The measurements and observation from the experiments help in confirming, rejecting, or refining the hypothesis. The experiments performed can be either physical or computational. Computational experiments, i.e., simulations, are typically used for one of two reasons. First, they can help in saving time and/or costs compared to performing physical experiments. Second, in some cases, they can model natural phenomena for which physical experimentation is impossible.

Computational simulations can take many forms. For many physics-based simulations, a three-dimensional volume (or two-dimensional area) is simulated using a "mesh" that discretizes the volume into many elements (such as hexahedra, tetrahedra, etc.). Physical quantities (such as pressure, temperature, or velocity) are represented as variables over these meshes.

Computational simulations must be accurate to be useful, and often supercomputers are needed to achieve this accuracy. The accuracy of a simulation depends on the underlying mesh and can be improved by increasing the number of elements in the mesh. Typically, a simulation cannot produce an accurate result unless the number of elements crosses some threshold, after which additional elements are no longer useful. Since the volume being simulated is often fixed in size, increasing the number of elements results in each element becoming “finer,” i.e., smaller in size (at least on average). That said, when the mesh resolution becomes very fine, then a single computer is often insufficient — it does not have enough memory to store the mesh nor enough computational power to carry out the required numerical operations (which increase as mesh size increases) in a reasonable amount of time. This issue is the main motivation for using supercomputers, as they have more memory and more computational power. As the requirements for memory and computational power become more extreme, the supercomputer must become larger and larger to keep pace. Currently, leading-edge simulations can require trillions of elements, each with many physical quantities. Such simulations serve as the motivation for exascale computing.

1.1.1 Exascale Computing.

The last decade has seen a shift from petascale computing (10^{15} FLOPS) to exascale computing (10^{18} FLOPS), i.e., to being capable of executing a billion billion floating point operations (1 exaFLOP) per second. This increase will in turn enable undertaking large scientific challenges. This shift has mainly been driven by advances in accelerator hardware, primarily Graphics Processing Units (GPUs), which are able to offer significant increases in FLOPS while also being energy efficient. Seven of the top ten most powerful supercomputers and nine of the top ten green supercomputers are GPU-based systems [4, 2].

However, even if the move towards exascale promises more computational power, using this computational power efficiently is a significant challenge. Developing code for accelerators like GPUs requires expertise with specialized tools and libraries like CUDA, HIP, etc. Accelerator hardware also works in a fundamentally different way than traditional CPUs, which further complicates optimizing applications. To add to that, a leading supercomputer has massive parallelism — thousands of nodes and multiple accelerator devices per node. As a result, developers will need to write programs that can benefit from billion-way concurrency.

The pursuit of exascale has also created a diverse ecosystem of computing hardware that requires the usage of specialized software stacks (compilers, profilers, etc.). This creates a challenge where a software implementation optimized for a specific platform might not perform as efficiently on other platforms creating a problem of portability for simulation and analysis software. However, to help with this problem, parallelization libraries like OpenMP, DPC++, Kokkos [45], RAJA [71], and VTK-m [95] are being developed to enable developers to write portable platform solutions.

Exascale computing is expected to enable advances in many fields of scientific computing. That said, not only will the computational simulations need to adapt to the nature of these computers, but also the software that works with these simulations. In particular, scientific visualization, which enables insight into simulation results, will need to adapt to the changing trends in computing.

1.1.2 Large Scale Visualization.

Visualization algorithms are commonly used to analyze the data output from simulations, since the imagery they produce often provides insight into key phenomena. The massive size of the data sets produced by large-scale simulations present the same issues for visualization as they do for the simulations. Visualization programs often take a similar approach to simulations, i.e., running in parallel on a supercomputer and dividing the data over its nodes. This process is simplified in two key ways: (1) the simulation typically already decomposes its mesh into "blocks" and these blocks can be used for parallel visualization and (2) a supercomputer is readily available to run parallel visualization, i.e., the same supercomputer that ran the simulation can be used for visualization. Traditionally, visualization has been performed “post hoc,” i.e., it occurs after a simulations has written its output to the disk, and it operates by reading data from disk. However, trends in supercomputing are challenging this paradigm.

While billion-way concurrency is one of the main challenges with exascale computing, a separate and potentially equally difficult challenge is with I/O bandwidth. Using accelerators like GPUs enable simulations to produce large volumes of data quickly. However, this increased rate of production is not being met by a commensurate increased rate in I/O bandwidth — I/O bandwidth is generally increasing on supercomputers, but much less than the ability to compute new data. As a result, the I/O bandwidth is going down relative to the computational power, making the traditional post hoc model of visualization increasingly infeasible. In response, large-scale simulations are increasingly using in situ visualization and analysis. In situ processing refers to visualizing or analyzing the simulation data as it is generated, and then saving the imagery or analyses instead of the simulation data. Importantly, this model requires visualization algorithms to not only run on accelerators, but also to not place undue burden on the simulation. In other words, it must execute quickly (typically by efficiently using all available parallelism) and not consume too much memory.

There are many visualization techniques that are applied to study simulations at a large scale. The diversity of visualization techniques and the different ways to achieve them complicates the goal to achieve efficiency. The focus of this dissertation are the visualization algorithms that enable study of vector fields representing some flow, also known as flow visualization. In particular, it focuses on flow visualization algorithms that use a technique called "particle advection." Since particle advection is of critical importance in the analysis of flow simulations, understanding and improving its performance is an important research problem, and serves as the main motivation for this thesis.

1.1.3 Flow Visualization.

Flow visualization is the branch of scientific visualization devoted to understanding flow fields, i.e., fields representing motion within a volume. It is used to study diverse scientific domains, such as climate change, ocean movement, and combustion. It helps to identify patterns in a flow and extract various qualitative or quantitative information about it. There are two broad techniques to visualize flows: experimental and computational. In experimental flow visualization, information is extracted by observing natural phenomena using captured pictures or direct observation. In computational flow visualization, simulations representing natural phenomena are performed, and the data is visualized using specialized algorithms and computer graphics. Computational flow visualization is usually performed using one of two techniques: particle advection and texture advection. This dissertation proposal focuses on computational flow visualization techniques that use particle advection, which is far and away the most commonly used approach. Particle advection involves tracing a particle's path in a flow, typically represented by a vector field, by solving ordinary differential equations (ODEs). The path is traced by taking a series of steps, also known as advection steps.

As flow simulations scale to tackle bigger problems, the computational requirements for particle advection-based algorithms often increase proportionally. That said, different particle advection-based algorithms have very different workload requirements. Some algorithms require few particles that travel for long durations, while others require many particles that travel for shorter durations. Further, some workloads require calculating billions of total advection steps, whether from many particles, from many steps per particle, or from both. Some algorithms also require additional analysis to be performed for every advection step. These factors affect the behavior of flow visualization algorithms and complicate their study.

These diverse flow visualization techniques and their corresponding workloads also exhibit differences in performance for the underlying particle advection components. The components constitute of different ODE solvers, different cell locators, different interpolation techniques, etc. The choice of these components has a substantial effect on the performance of an algorithm.

There is also a significant difference in the way vector fields are represented for different use cases. Most commonly, the flow is represented using a velocity field, and the particles being advected have no associated properties. However, in the case of electromagnetic fields, the flow is represented using electric and magnetic fields, and the particle velocities are calculated using the mass and charge on the particle. Implementing a general system that can handle both the typical case of a vector field and more exotic cases (such as the electromagnetic fields just discussed) introduces another set of challenges.

Apart from the diverse workloads, components, and data representations, the performance of particle advection is also determined by how efficiently it can use its execution environment. In the context of a single node, particles can be considered as individual work units for parallelization. When particle advection

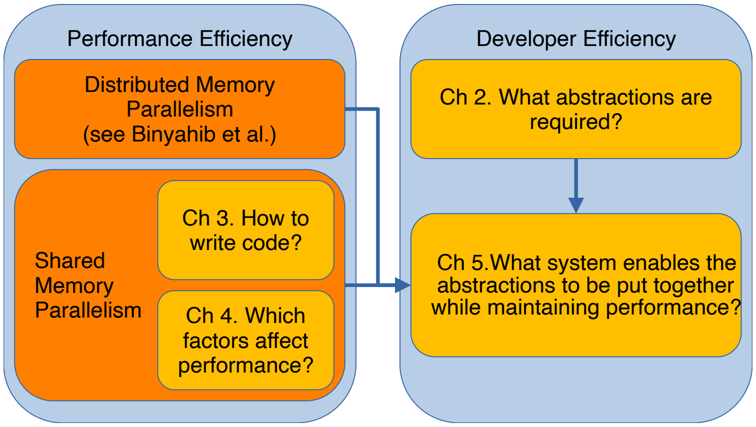

Performance Efficiency
Distributed Memory Parallelism (see Binyahib et al.)
Shared Memory Parallelism
Ch 3. How to write code?
Ch 4. Which factors affect performance?
Developer Efficiency
Ch 2. What abstractions are required?
Ch 5.What system enables the abstractions to be put together while maintaining performance?

Figure 1. Figure describing the organization of chapters towards answering the dissertation question. The blue boxes represent the two different types of efficiencies that are describes in the system, the orange boxes represent the two different types of parallelisms that are required for an efficient system, and finally, the yellow boxes represent the research presented in this dissertation. The arrows represent the relation of the different components towards designing the general purpose system.

workloads are large, more concurrency is available to parallelize the work. However, the efficient use of parallelism is challenging and requires more research.

1.1.4 Thesis Question. The central question for my dissertation is in response to the challenges mentioned above: "What flow visualization system designs will enable both efficiency on exascale systems and be capable of supporting diverse analysis needs?" The question can be further broken down to achieve two different objectives:

- What methods and approaches will enable efficient performance on exascale machines?
- What system design can both address diverse analysis needs while also delivering performance on exascale machines?

Section 1.2 describes the strategy to answer the first question, while Section 1.3 describes the strategy to answer the second question. Further, figure

1 describes the organization of the chapters in this dissertation that answer the posed dissertation question.

For performance efficiency, using parallelism is of paramount in the age of exascale computing. For shared memory parallelism, Chapter III describes the approaches for designing a platform portable particle advection system, while Chapter IV describes the factors that impact the performance of particle advection in a shared memory environment and also suggests best practices for achieving the best performance which goes beyond conventional wisdom. For distributed memory parallelism, my thesis question benefits from previous research, namely the dissertation by Roba Binyahib [15] which was focused on distributed-memory performance. Since Binyahib's thesis is very recent and highly applicable, my dissertation does not contribute any research towards distributed memory parallelism.

For developer efficiency, providing useful abstractions and means to extend them without much effort are key. Chapter II identifies the different components of the particle advection algorithm and details the different optimizations that are applied to each of them. These components become the basis for designing abstractions for the general purpose system proposed in Chapter V. Further, the research for Chapter III and the research by Binyahib et al. are important to support the abstraction for using the exascale hardware efficiently, making it easier to adapt to newer and more diverse computing resources. Finally, the takeaways from Chapter IV about the interplay of various factors related to the particle advection workload and their impact on performance should help visualization practitioners to make informed choices about using the system more efficiently.

The relationship between all the chapters towards the system proposed in Chapter V are represented by arrows in Figure 1.

1.2 Performant Particle Advection System

This section proposes research to better understand the performance of particle advection. In turn, this understanding can then be used to optimize performance. Studying particle advection performance is important for flow visualization algorithms in general; while these algorithms have aspects that do not use advection, the computational cost of an algorithm is often dominated by the advection portion. We organize the factors that affect particle advection performance into two types:

Intrinsic: defined as “belonging naturally; essential,” these are the factors that are hardware agnostic, i.e., they do not have a particular relation to the execution environment of the algorithms. Changes made to these factors have the same behavior across all platforms.

Extrinsic: defined as “not part of the essential nature of someone or something; coming or operating from outside,” these are the factors that are hardware dependent, i.e, they have a relation to the execution environment of the algorithms. Changes made to these factors result in different behaviors on different platforms.

This dissertation proposal outlines studies to understand the performance impacts for both of these types of factors. Section 1.2.1 proposes research towards understanding particle advection as an algorithm, i.e., intrinsic. Section 1.2.2 proposes research towards understanding the behavior of particle advection and optimizing it for various execution environments, i.e., extrinsic.

1.2.1 Studies for Intrinsic Factors. Particle advection performance is complicated. The algorithm appears to be (and sometimes is) straightforward and embarrassingly parallel: particles are treated as individual work units and advanced independently. However, simply adding more concurrency to parallelize the workload does not always lead to proportional speedups. Even advecting a single particle can be challenging due to the performance issues introduced by various components responsible for the algorithm. The process of particle advection requires evaluation of the particle's velocity and then solving an ordinary differential equation (ODE). Each of these operations involves a computation. The particle's next position (i.e., the outcome of a single advection step) is calculated by the ODE solver using the particle's velocity. Further, depending on the nature of the ODE solver being used, multiple velocity evaluations are usually needed at locations near the particle. For evaluating the velocity at a location, the cell containing the location ("containing cell") in the mesh is determined using a cell location data structure. Then the velocity is interpolated using this cell's information. The process of searching the cell for a location, gathering its velocities, and interpolating the velocities for the location involves expensive memory accesses. Understanding the impact of these operations on particle advection as a whole is critical for performance improvements. For the topic of intrinsic factors, I propose one research direction, described in Section 1.2.1.1. This section summarizes existing research for applying specific optimizations to the individual components of particle advection and makes a new contribution in providing a framework for assessing the potential speedup that can be realized using such optimizations.

1.2.1.1 Algorithmic Optimizations and Cost Modeling. This study proposes the identification and organization of all the components of particle advection that part take in completing the workload for a given flow visualization algorithm. A workload, in this case, is termed as the total number of advection steps for a given algorithm (number of particles × number of steps). This study aims to enable users to make optimization decisions for particle advection based on the cost of the workload and the budget of allowed execution time. The study involves formulating a high-level analytical cost function that can determine the number of FLOPs necessary to execute the workload with a hope that this cost is proportional to the execution time for the algorithm. Our meta-study will survey various studies that propose optimizations to individual components of particle advection (e.g., cell locators, ODE solvers) and the amount of speedup these studies can achieve. Visualization scientists can use these techniques and the speedups to decide which optimizations to apply based on calculated costs. The final result is a decision-making workflow that users can use to choose algorithmic features for their workload. The outcome of the preliminary work towards this topic is discussed in Chapter II.

1.2.2 Studies for Extrinsic Factors. The performance of particle advection is heavily affected by the execution environment. Some of the factors that directly contribute to the performance include the accelerator hardware, the number of threads, and cache sizes. These factors can affect the scheduling of particles if insufficient concurrency is available or stress the cache hierarchy introducing bottlenecks. The extent to which these factors affect the workload also depends on the workload characteristics, namely the number of particles, the duration of advection, the initial placement of the particles, and the nature of the vector field. Hence, understanding the impact of the execution environment combined with the workload is essential to determine efficient optimizations. This is particularly true for the exascale computers as these are heterogeneous systems offering both many-core CPUs and diverse accelerator hardware. The studies proposed in this section are aimed to understand execution environment effects.

1.2.2.1 Performance Portability. The heterogeneous nature of exascale systems and the diversity in the accelerator hardware (Nvidia GPUs, AMD GPUs, Intel GPUs) create many distinct execution environments. This creates a problem for developers of scientific tools where optimizations applied to one of the execution environments might not work well on the others. It is also not feasible to write algorithms that are specifically optimized for each of these platforms, as it drives down the maintainability of software and decreases developer efficiency. The scientific visualization community developed the VTK-m library to address this problem. VTK-m uses data parallel primitives as building blocks for visualization algorithms [95]. The library maintains optimized versions of the data parallel primitives, and the developers express their operations using them as building blocks. Developers can thus produce algorithms that can perform comparably to algorithms that are optimized specifically to a certain platform, making their algorithms "platform-portable." The objective of this study was to research approaches for platform portable particle advection within a data-parallel environment. In terms of status, we developed a working, efficient algorithm, and published a paper describing its details and performance results [110]. The paper compared the implementation with platform-specific, widely used comparators and found that our approach can offer great performance portability. The implementation is now a part of the VTK-m library. The results of this work are discussed in greater depth in Chapter III.

1.2.2.2 Particle Advection Speedups from GPU and CPU Parallelism. The particle advection implementation in VTK-m demonstrated

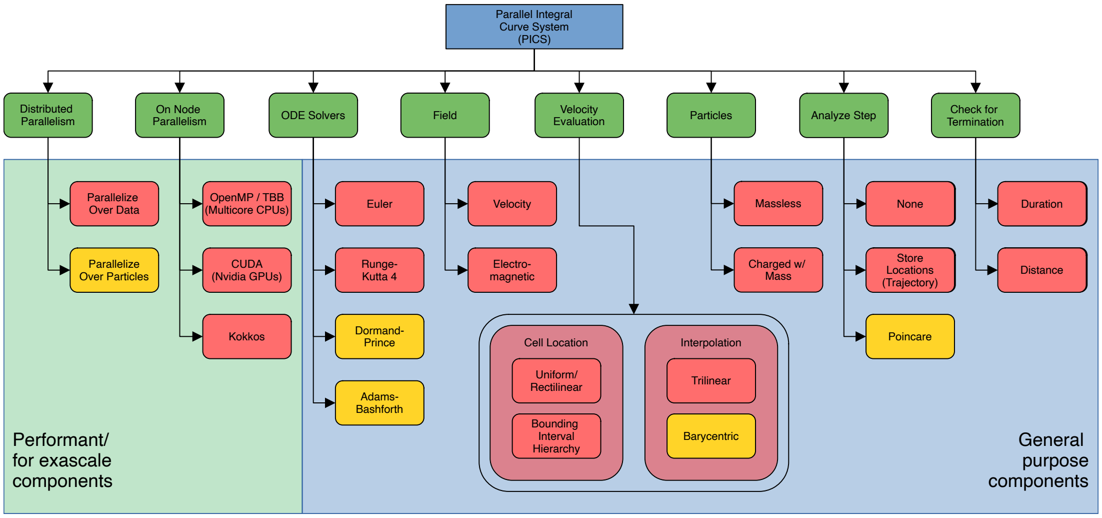


Figure 2. The organizations of a general purpose flow visualization system for exascale computers. The components encapsulated in the green box are the components that make the system exascale ready. The components encapsulated in the blue box are the components that make the system extendible and general purpose. Together these components achieve the goal of "efficiency squared," aiming for both performance efficiency and developer efficiency.

good portability against its parallel comparators [110]. The study was performed using two generations of GPUs and demonstrated massive differences in their speedups. With each new generation, multi-core CPUs and GPUs make a leap in their performance capabilities. This makes the projecting the potential speedups for particle advection using these execution devices difficult, and this is the key challenge this chapter seeks to address. This study aims to identify trends of performance improvement between different generations of GPUs and multi-core CPUs for particle advection. It considers different workloads, different data sets, and different visualization algorithms, along with different execution devices to understand the performance impact of each of these factors. Based on these different parameters, this study points out key takeaways about particle advection and its performance: (a) it compares the speedups of GPUs against serial execution and identifies the impact of GPU generations, (b) it compares the speedups of multi-core CPUs against serial execution and identifies the impact of available concurrency, (c) it identifies cases where developers should favor serial or multi-core CPUs over GPUs (d) it investigates the impact of the data set characteristics on particle advection performance. The results of this work are discussed in greater depth in Chapter IV.

1.3 General Purpose Flow Visualization System for Exascale

There is a wide diversity in the needs and systems to meet flow visualization demands. Research is needed to unify these needs into a single system by providing abstractions for various components for flow visualization and providing concrete types for those abstractions. The components can later be used or extended across algorithms and user groups. A unified system will result in an extensible system that can address the diverse needs of flow visualization rather than make repeated investments to develop new systems. Saying it another way, this system has the potential to provide significant savings in developer time by providing a single system where otherwise many would be needed.

To that end, we propose a flow visualization system that addresses two essential requirements for a unified system:

Performance Efficiency: The system can efficiently use exascale computing resources, and in particular both distributed-memory and shared-memory hardware.

Developer Efficiency: The system readily provides necessary abstractions for the diverse visualization and analysis demands of different algorithms, user groups, and data representations. If not, the system is extensible such that users can specify custom components without much effort.

We call achieving these efficiencies "efficiency squared," since the goal is to achieve developer efficiency without sacrificing the system's performance efficiency.

Pursuing both of thes goals simultaneously makes designing and implementing this system challenging. i.e., achieving these efficiency goals in isolation, while not trivial, is simpler than considering both of them together.

To that end, this study aims to design and implement a flow visualization system that satisfies the "efficiency squared" criteria. Figure 22 describes its organization. The system makes provisions to accommodate custom specifications of components and analyses to make the system extensible. All the components represented with a green rectangle in the organization present the abstractions necessary for the system, which users can extend and customize.

The goal of performance efficiency is achieved by the components encapsulated in the larger green box. The implementation will perform shared memory parallelism using VTK-m as described in Chapter III.

The goal of developer efficiency is achieved by the components encapsulated in the larger blue box. The system will provide components that can be readily used to specify popular visualization algorithms like streamlines or Finite Time Lyapunov Exponents (FTLE). The system will provide flexibility and extendibility such that users can design their custom algorithms easily.

To study the efficacy of the implemented system, we aim to present case studies demonstrating exotic flow visualization use cases from different scientific domains. These case studies will utilize diverse components of the system defined in Figure 1. Implementing these visualization techniques and studying their performance should evaluate the "efficiency squared" criteria. The results of this work are discussed in greater depth in Chapter V.

CHAPTER II

BACKGROUND

This chapter is an extended version of my area exam (candidacy exam). It is currently in submission. I was the primary author for the original areaa exam manuscript, with editorial suggestions from Hank Childs. As this work was transformed into a survey paper, several authors made contributions: Sudhanshu Sane to the "Precomputation" section, Christoph Garth to the "ODE solvers" section, and David Pugmire and Roba Binyahib to the "Hardware Efficiency" section.

The performance of particle advection-based flow visualization techniques is complex, since computational work can vary based on many factors, including number of particles, duration, and mesh type. Further, while many approaches have been introduced to optimize performance, the efficacy of a given approach can be similarly complex. In this chapter, we seek to establish a guide for particle advection performance by conducting a comprehensive survey of the area. We begin by identifying the building blocks for particle advection and establishing a simple cost model incorporating these building blocks. We then survey existing optimizations for particle advection, using two high-level categories: algorithmic optimizations and hardware efficiency. The sub-categories of algorithmic optimizations include solvers, cell locators, I/O efficiency, and precomputation, while the sub-categories of hardware efficiency all involve parallelism: shared-memory, distributed-memory, and hybrid. Finally, we conclude the survey by identifying current gaps in particle advection performance, and in particular on achieving a workflow for predicting performance under various optimizations.

2.1 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 of the trajectories are 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. These workloads consist of many factors, including the number of particles, duration of advection, velocity field evaluation, and analysis needed for each advection step. One particularly important aspect with respect to performance is the number of advection steps, which derive from both the number of particles and their durations. Many flow visualization techniques have numerous particles that go for short durations, while many others have few particles that go for long durations. Some cases, like when analyzing flow in the ocean [101], require numerous particles for long durations and thus billions of advection steps (or more). With respect to 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 makes it difficult to reason about both the execution time for a given workload and the potential improvement from a given optimization.

The main goal of this chapter is to provide a guide for understanding particle advection performance, including possible optimizations. It does this in four parts. First, Section 2.2 provides background on the building blocks for particle advection. Second, Section 2.3 introduces a cost model for particle advection performance, to assist with reasoning about overall execution time and inform which aspects dominate runtime. Third, Section 2.4 surveys algorithmic optimizations. Fourth, Section 2.5 surveys approaches for utilizing hardware more efficiently, with nearly all of these works utilizing parallelism. Contrasting the latter two sections, Section 2.4 is about reducing the amount of work to perform, while Section 2.5 is about executing a fixed amount of work more quickly.

In terms of placing this survey into context with previously published literature, we feel this is the first effort to provide a guide to particle advection performance. The closest work to our own is the survey on distributed-memory parallel particle advection by Zhang and Yuan [137]. Our survey is differentiated in two main ways. First, our survey considers a broader context overall, i.e., it considers algorithmic optimizations and additional types of parallelism. Second, our discussion of distributed-memory techniques does a new summarization of workloads and parallel characteristics (specifically Table 5), and also has been updated to include works appearing since their publication. There also have been many other excellent surveys involving flow visualization and particle advection: feature extraction and tracking [106], dense and texture-based techniques [82], topology-based flow techniques [83] and a subsequent survey focusing on topology for unsteady flow [104], integration-based, geometric flow visualization [91], and seed placement and streamline selection [115]. Our survey complements these existing surveys — while some of these works consider aspects of performance within their individual focal point, none of the surveys endeavor to provide a guide to particle advection performance.

2.2 Particle Advection Background

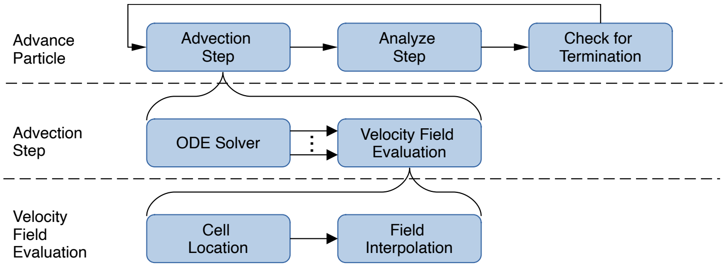


Figure 3. Organization of the components for a particle advection-based flow visualization algorithm. The components are arranged in three rows in decreasing levels of granularity from top to bottom. In other words, the components at the bottom are building blocks for the components at higher levels. 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.

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 forms they happen in an overlapping fashion (i.e., seed, advance, seed more, advance more, etc.)

Our organization, which is illustrated in Figure 3, 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 to its next position.
- ***Analyze Step:*** analyzes the advection step that was just taken.
- ***Check for Termination:*** determines whether a particle should be terminated.

The process of advancing a particle involves three phases that are applied repeatedly. The first phase is to displace a particle from its current location to a new location. Such displacements are referred to as particle advection steps. The second phase is to analyze a step. The specifics of the analysis vary by flow visualization algorithm, and could be as simple as storing the particle's new location in memory or could involve more computation. The third phase is to check if the particle meets the termination criteria. Similar to the second phase, flow visualization algorithms define specific criteria for when to terminate a particle. Finally, if the particle is not terminated, then these three phases are repeated in a loop until the termination criteria are reached.

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

- ***ODE Solver***: calculates a particle's displacement to a new position by solving an ordinary diffential equation (ODE).

- ***Velocity Field Evaluation:*** calculates the velocity value at a specific location by interpolating within the located cell.

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

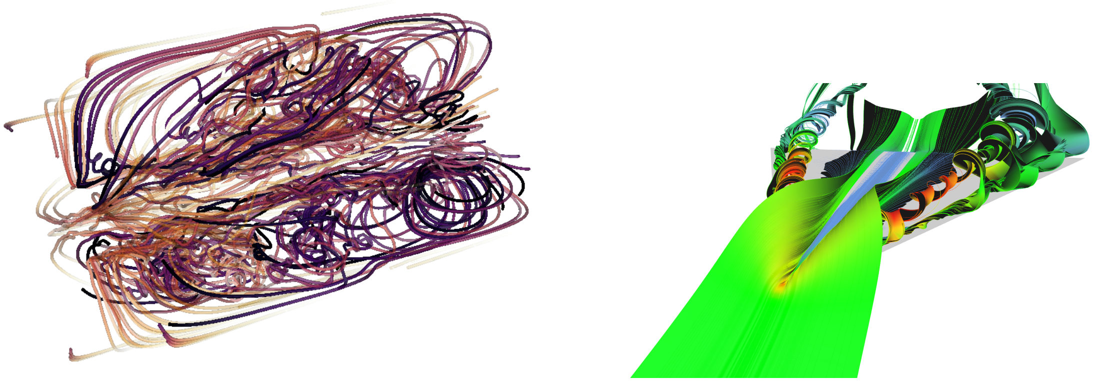
The “bottom” level of our organization considers the process of velocity field evaluation. This level also has two components:

- ***Cell Location:*** locates the cell that contains some location.

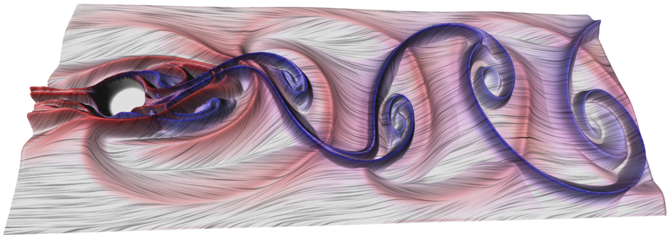
- ***Field Interpolation:*** calculates velocity field at a specific location via interpolation of surrounding velocity values.

In terms of a relationship, to calculate velocity at some point P, first cell location is used to identify the cell C that contains P, and then velocity field interpolation is used to calculate the velocity at P using C’s information.

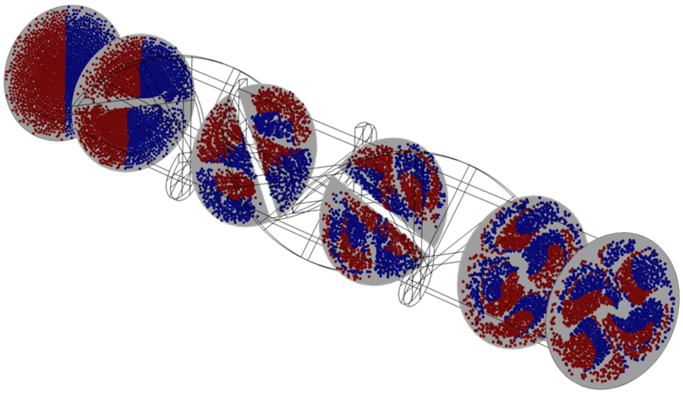
Different flow visualization algorithms use these components in different ways, resulting in different performance across the algorithms. One way of comparing the different algorithms’ perforamance can be the workload required by the algorithms, which roughly translates to the total computation required by the 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 4 shows examples of four different flow visualization algorithms that demonstrate significant differences in their workloads and behaviours. Table 1 highlights the differences between the workloads for the example algorithms.



(a) streamlines (b) streamsurface



(c) FTLE



(d) Poincaré

Figure 4. 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 [105], subfigure (b) shows a streamsurface which is split by turbulence and vortices that can be observed towards the end [51], subfigure (c) shows attracting (blue) and repelling (red) Lagrangian structures extracted as FTLE ridges from a simulation of a von Korman vortex street [74], 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 [86].

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 catagories.

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

(b) Typical parameter configurations for different flow visualization algorithms.

2.3 Cost Model for Particle Advection Performance

This section considers costs from the perspective of the building blocks used to carry out particle advection. Its purpose is to build a general framework for reasoning about costs and also to inform which aspects contribute most to overall cost. That said, the simplicity of the cost model precludes directly evaluating many of the optimizations described in later sections (I/O, parallelism, precomputation, and adaptive step sizing); this topic is revisited in Section 2.6. Finally, Appendix A goes into more depth on the cost model, including estimating costs for each term in the model, notional examples, and validation of the model.

Let the costs for particle advection be denoted by $Cost$. Then a coarse formulation for $Cost$ is:

$$Cost = \sum_{i=0}^{i=P} \sum_{j=0}^{j=N_i} advance_{i,j} \tag{2.1}$$

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

To better illuminate the overall costs, the remainder of this section considers how the coarse formulation in Equation 2.1 can be further decomposed. We first consider the tasks within $advance_{i,j}$. In particular, each step that advances a particle contains three components — taking an advection step, analyzing the step in a way specific to the individual flow visualization algorithm, and checking if the particle should be terminated. Hence, Equation 2.1 can be written as:

$$Cost = \sum_{i=0}^{i=P} \sum_{j=0}^{j=N_i} \Big(step_{i,j} + analyze_{i,j} + term_{i,j} \Big) \tag{2.2}$$

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

The cost can be further broken down by exploring the cost for a single advection step, $step_{i,j}$. Particle advection uses an ODE solver to determine the next position of a particle, and this solver requires the velocity of the particle at the current location. Further, depending on the ODE solver, additional velocity evaluations in the proximity of the particle may be required. An Euler solver requires only one velocity evaluation, while an RK4 solver requires four velocity evaluations. Generalizing, the cost of a single particle advection step can be written as:

$$step_{i,j} = solve_{i,j} + \sum_{k=0}^{k=K} eval_{i,j,k} \tag{2.3}$$

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

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

$$eval_{i,j,k} = locate_{i,j,k} + interp_{i,j,k} \tag{2.4}$$

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

Further, we can substitute 2.4 in 2.3 to yield:

$$step_{i,j} = solve_{i,j} + \sum_{k=0}^{k=K} (locate_{i,j,k} + interp_{i,j,k}) \tag{2.5}$$

Finally, we can substitute 2.5 in 2.2 to obtain our final formulation:

$$\begin{aligned} Cost = \sum_{i=0}^{i=P} \sum_{j=0}^{j=N_i} \Big(solve_{i,j} + \sum_{k=0}^{k=K} \big(locate_{i,j,k} + interp_{i,j,k} \big) \\ + analyze_{i,j} + term_{i,j} \Big) \end{aligned} \tag{2.6}$$

2.4 Algorithmic Optimizations

This section surveys algorithmic optimizations for 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 2.4.1 discusses optimizations to ODE solvers, Section 2.4.2 discusses optiizations for cell location, Section 2.4.3 discusses strategies to improve I/O efficiency, and finally Section 2.4.4 discusses strategies that involve precomputation.

2.4.1 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, and we refer the reader to the excellent book by Hairer et al. [64] 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 error of the approximation of the solution, which is achieved by automated selection of the step size (which in turn controls the approximation error) in each step. Often used methods in this context are the Runge-Kutta Fehlberg (RFK) method [64], the Runge-Kutta Cash-Karp (RKCK) method [64], and the Dormand-Prince fifth-order scheme (DOPRI5) [107]. As an additional benefit relevant in this context, due to the typically low error tolerances required for visualization purposes, significant performance benefits can be obtained if the adaptive step sizing results in fewer large steps taken compared to fixed-step size methods. 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 [57, 66].

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. [114] 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.

2.4.2 Cell Locators. Cell locators facilitate interpolation queries over a grid and rely on auxiliary data structures that partition candidate cells spatially. These are typically constructed in a pre-processing step and induce linear memory overhead in the number of cells N, while accelerating queries to $\mathcal{O}(\log N)$. 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. According to Lohner and Ambrosiano [89] the process of cell location can follow one of the following three approaches.

Using a Cartesian background grid: Cell 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: A basic approach to hierarchical cell location is the use of octrees for cell location [120, 132]. 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.[80], based on a kd-tree storing just the vertices of an unstructured grid. This allows to 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* [54], 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 Tricoche [8] 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 accommodates large meshes with significant variations in cell shapes and sizes well. While Lohner and Ambrosiano 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 [54].

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, as well as Ueng et al. [125], 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*** [118, 28].

Kenwright and Lane [77] extended the work by Ueng et al. by improving the technique to identify the particle's containing tetrahedron. Their approach uses fewer floating point operations for cell location compared to Ueng et al.

Successive neighbor search is also naturally incorporated in the method of Langbein et al. [80]; ray casting with adjacency walking towards begins at the previous interpolation point in this case.

2.4.3 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 of computation is known as out-of-core computation. One of the significant bottlenecks for flow visualization algorithms while performing out-of-core computations is the cost of I/O. Particle advection is a data-dependent operation and efficient prefetching to ensure sequential access to data can be very beneficial in minimizing these I/O costs. This section discusses the works that aim to improve particle advection performance by improving the the efficiency of I/O operations.

Chen et al. [36] 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 speedups over the Hilbert curve layout [69].

Chen et al. [34] extended the previous work to out-of-core computation of pathlines. Their results show a performance improvement in the range of 10%-40% compared to the Z-curve layout [138, 134].

Chen et al.[35] expanded the work further to introduce a seed scheduling strategy to be used along with the graph-based data layout. They demonstrated an efficient out-of-core approach for calculating FTLE. The performance improvements observed against the Z-curve layout were in the range of 8%-32%.

2.4.4 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, based on current literature, a set of particle trajectories can be computed to inform data access patterns, or serve as a basis for interpolating new trajectories. Depending on the objectives, the number of trajectories computed during the first stage varies. The resulting set of trajectories can be referred to as the precomputed trajectories.

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. [36] in the previous section. A similar approach was studied by Nouansengsy et al. [100] to improve load balancing in a distributed memory setting. In these cases, the first stage is a preprocessing step and a small number of particle might be advected to form the set of precomputed trajectories.

For a computationally expensive particle advection workload, a strategy to accelerate the computation or improve interactivity of time-varying vector field visualization is to divide the workload into two sets. The first set includes particle trajectories computed using high-order numerical integration. The second set includes particle trajectories that are derived by interpolating the precomputed trajectories. If new particle trajectories can be derived from the precomputed set faster than numerical integration, while remaining accurate and satisfying particle trajectory requirements for the specific flow visualization use case, then the total computational cost of the workload can be reduced compared to the numerical intergration of every trajectory.

Hlawatsch et al. [70] 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 the finite-time Lyapunov exponent and the line integral convolution. Although the method introduces a trade-off of 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. [20] 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. [20] 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. [5] 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, can support 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 accuracy-storage propositions. For example, Sane et al. [116] studied the use of longer trajectories to reduce error propagation and improve accuracy, and Rapp et al. [111] proposed a statistical sampling technique to determine where seeds should be placed. Unstructured sampling strategies, however, can increase the cost of post hoc interpolation and diminish computational performance benefits.

2.4.5 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 optimizations 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 speedups for the workloads. With a

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
[89]	Streamlines	Fast cell location and efficient vectorization	870^*	-	10K	14×
[125]	Streamlines	Streamline computation and cell location in canonical coordinate space	$320K^*$ $225K^*$ $288K^*$	- - -	100	1.61× 1.59× 1.58×
[36]	Streamlines	Improving data layout for better I/O performance	134M 200M 537M	- - -	4K	$0.96 - 1.30\times$ $0.98 - 1.98\times$ $0.99 - 1.29\times$
[34]	Pathlines	Improving data layout for better I/O performance	25M 65M 80M	48 29 25	4K	$1.25 - 1.38\times$ $1.10 - 1.31\times$ $1.19 - 1.36\times$
[35]	FTLE	Improving data layout for better I/O performance	25M 65M 80M	48 29 25	-	$1.08 - 1.32\times$

combination of efficient cell location and vectorization, Lohner and Ambrosiano [89] achieved the speed of 14×. However, the other study [125] demonstrated a speedup of around 1.6×. The works by Chen et al. [36, 34, 35] for efficient I/O for particle advection all demonstrated speedups up to 1.3×.

2.5 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. This means 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 on along with multiple accelerators (e.g., GPUs) for parallelization. As a result, algorithms are expected to make efficient use of this billion-way concurrency. This section discusses research for particle advection that addresses efficient usage of available hardware. Section 2.5.1 discusses research that aims to improve shared-memory (on-node) parallelism. Section 2.5.2 discusses research that aims to improve distributed memory parallelism. Section 2.5.3 discusses research that uses both shared and distributed memory parallelism.

2.5.1 Shared Memory Parallelism for Particle Advection.

Shared memory parallelism refers to using parallel resources on a single node. The devices that enable shared memory parallelism are multi- and many-core CPUs and other accelerators, such as GPUs. In the case of shared memory parallelism, multiple threads of a program running on different cores of a processor (CPU or a

GPU) share memory, hence the nomenclature. One of the primary reasons for the increase in supercomputers' compute power can be attributed to the advancements of CPUs and accelerator hardware. In all, for applications to make cost-effective use of resources, it has become exceedingly important to use shared memory resources efficiently. However, making efficient use creates many challenges for the programmers and users. Two important factors to consider are 1) efficient use of 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 [4]. Part of this has been the availability of specialized toolkits, including early efforts like Brook-GPU [23] and popular efforts like Nvidia's CUDA [98], that enable GPUs to be used as general purpose computing devices [13]. 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. That said, published research to date has limitations with respect to understanding how much benefit parallelism provides. We address this limitation with our work in Chapter IV. Section 2.5.1.1 discuss works to optimize the performance particle advection on GPUs and Section 2.5.1.1 describes optimizations for cell locators. Section 2.5.1.2 discusses works that use CPUs for improving the performance of particle advection.

2.5.1.1 Shared memory GPUs. Most of the solutions that focused on shared memory optimization focused on improving the performance on GPUs. This is because particle advection can benefit from using GPUs when there are many particles to advect. As particles can be advected independently from one another, each particle can be scheduled with a separate thread of the GPU, making the most of the available concurrency. Many works have tried to address performance issues of particle advection using GPUs, however, with different goals.

Krüger et al. [78] presented an approach for interactive visualization of particles in a steady flow field using a GPU. They exploited the GPU's ability to simultaneously perform advection and render results without moving the data between the CPU and the CPU. This was done by accessing the texture maps in the GPU's vertex units and writing the advection result. Their approach on the GPU provided the interactive rendering at 41 fps (frames per second) compared to 0.5 fps on the CPU.

Bürger et al. [27] 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. [26] 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. [24] also provided a way for interactively rendering streak surfaces. Using GPUs, the streak surfaces can be adaptively refined/coarsened while still maintaining interactivity.

Cell Locators for GPUs Bußler et al. [28] presented a GPU-based tetrahedral walk for particle advection. Their approach for cell location borrowed heavily from the work by Schiriski et al. [118] discussed in Section 2.4.2. 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 that the ***single-pass*** method, which performs only one pass through the kd-tree to find the nearest cell vertex (without the guarantee of it being the nearest) performs the best. The other strategies evaluated in the study were ***random restart*** and ***backtracking***.

Garth and Joy [54] presented an approach for cell location based on bounding interval hierarchies. Their search structure, called ***celltree***, improves construction times via a heuristic to determine good spatial partitions. The authors presented a use case of advecting a million particles on a GPU in an unstructured grid with roughly 23 million hexahedral elements. The celltree data structure was able to obtain good performance on GPUs despite no GPU-specific optimizations.

2.5.1.2 Shared Memory Parallelism on CPUs. Hentschel et al. [68] presented a solution that 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 that data that are required to perform the advection computation. This work demonstrated the advantage of packaging particles into spatially local groups where SIMD extensions are able to be more efficient. Their approach resulted in performance improvements of up to 5.6X over the baseline implementation.

Finally, Pugmire et al. [110] provided a platform portable solution for particle advection using the VTK-m library. The solution builds on data parallel primitives provided by VTK-m. Their results demonstrated very good platform portability, providing comparable performance to platform specific solutions on many-core CPUs and Nvidia GPUs.

2.5.1.3 Summary. In terms of published research, Table 3 presents a summary of shared memory particle advection. These studies either presented approaches for interactive flow visualization or optimizations for particle advection of GPUs using cell locators, with one exception that demonstrated platform portability.

The performance difference for particle advection between two generations of GPUs can be significant. Existing studies fail to capture this relation and makes it harder to estimate to speedup that can be realized. The research to estimate the speedups realized by using a newer GPU is discussed in Chapter IV.

2.5.2 Distributed Memory Parallelism for Particle Advection. Fluid simulations are capable of producing large volumes of data. Analyzing and visualizing volumes of data to extract useful information demands resources equivalent to that of the simulation. In most cases, this means access to many nodes of a supercomputer to handle the computational and memory needs of the analysis. Particle advection based flow visualization algorithms often execute in a

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
[78]	source-dest	Interactive flow visualization (steady) using GPUs	-	-	-	60-80×
[27]	various	Interactive flow visualization (unsteady)				
[26]	various	Interactive flow visualization using importance metrics	7M 4M 1M	- 22 30		
[25]	streak surface	Interactive streak surface visualization	589K 4.1M	102 22	400	
[118]	pathlines, source-dest	Efficient cell location on GPUs	$0.8M^*$ $1.1M^*$ $3.7M^*$	5 101 200	1M	
[54]	source-dest	Efficient cell location on GPUs for unstructured grids / Comparison against CPUs	$23.6M^*$	-	250K 1M	16.5×
[28]	source-dest	Efficient cell location on GPUs using improved tetrahedral walk	$4.2M^*$ $115M^*$ $743M^*$	5 101 200	1M	
[110]	source-dest	Performance Portability / Comparison with specialized comparators for CPUs and GPUs	134M 134M 134M	-	10M	$0.37 - 0.48\times$ (GPUs) $0.29 - 0.36\times$ (CPUs) $1.56 - 2.24\times$ (GPUs) $0.79 - 0.84\times$ (CPUs) $1.42 - 2.04\times$ (GPUs) $0.51 - 0.59\times$ (CPUs)

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

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

Most distributed particle advection solutions are either an optimization of these two classes or a combination of them. The decision to choose between these two classes depends on multiple factors, of which Camp et al. [32] identify the most prominent to be:

The volume of data set: 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.

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

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

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

This section describes distributed particle advection works in two parts. Section 2.5.2.1 describes the optimization for parallelizing distributed particle advection in more depth. Section 2.5.2.2 summarizes findings from the survey of distributed particle advection studies.

2.5.2.1 Parallelization Methods.

This section presents distributed particle advection works in three parts. Section 2.5.2.1 presents works that optimize parallelization over data. Section 2.5.2.1 presents works that optimize parallelization over particles. Section 2.5.2.1 presents works that use a combination of parallelization over data and particles.

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

Sujudi and Haimes [124] 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 request and sending streamlines across to the clients with the correct sub-domain. No details of the method used to decompose the data in sub-domains are provided.

Camp et al. [32] 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 [37]. 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. [133] 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 ensure workload balance. It also enabled them to render pathlines at different levels of abstraction.

Nouanesengsy et al. [100] used pre-processing to estimate the workload for each data block by advecting the initial set of particles. The estimates calculated from this step are used to distribute the work among processes. Their proposed solution maintained load balance and improved performance. While the solutions in this category are better at load balancing, they introduce an additional step of pre-processing which has its costs. This cost may be expensive and undesirable if the volume of data is significant.

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

Peterka et al. [102] performed a study to analyze the effects of data partitioning on the performance of particle tracing. Their study compared static round-robin (also known as block-cyclic) partitioning to dynamic geometric repartitioning. The study concluded that while static round-robin assignment provided good load balancing for random dense distribution of particles, it fails to provide load balancing when data blocks contain critical points. They also noticed that dynamic repartitioning based on workload could improve the execution time between 5% to 25%. However, the costs to perform the repartitioning are restrictive. They suggest more research needs to focus on using less synchronous communication and improvements in computational load balancing.

Nouanesengsy et al. [99] extended the work by Perterka et al. to develop a solution for calculating Finite-Time Lyapunov Exponents (FTLE) for large time-varying data. The major cost in performing FTLE calculations is incurred due to particle tracing. Along with *parallelize over data*, they also used *parallelize over time*, which enabled them to create a pipeline that could advect particles in multiple time intervals in parallel. Although their work did not focus on load-balancing among processes, it presented a novel way to optimize time-varying particle tracing. Their work solidifies the conclusions about static data partitioning of the study by Peterka et aL [102].

Zhang et al. [135] 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 processes without 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.

In conclusion, pre-processing works can achieve load balance with an additional cost for *parallelize over data.* This cost goes up with large volumes of data. The overall time for completing particle advection might not benefit from the additional cost of pre-processing, especially when the workload is not compute-intensive. Most solutions that rely on dynamic load balancing suffer from increased communication costs or are affected by the distribution of particles and the complexity of the vector field. The work proposed by Zhang et al. [135] is promising but still does not guarantee optimal load balancing.

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

Previous works have explored different approaches to optimize the scheme described above. Since the blocks of data are loaded whenever requested, the cost of I/O is a dominant factor in the total time. Prefetching of data involves predicting the next needed data block while continuing to advect particles in the current block to hide the I/O cost. Most commonly, predictions are made by observing the I/O access patterns. Rhodes et al. [112] used these access patterns as a priori knowledge for caching and prefetching to improve I/O performance dynamically. Akande et al. [7] 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 [36, 34, 35]. Guo et al. [61] used the access dependencies to produce fine-grained partitions that could be loaded at runtime for better efficiency of data accesses.

Zhang et al. [136] 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. [90] 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. [32] 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. [30] 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 [22, 44, 121]. In the case of integral curve calculation, task-based parallelism inspires the parallelize over particles scheme. Dinan et al. [44] demonstrated the scalability of the work-stealing approach. Lu et al. [90] 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 advancing algorithm proposed by Garth et al. [56], 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. [97] 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%.

Binyahib et al. [16] 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 partilces is a much better strategy and can result in speedups upto 10×.

According to Childs et al. [42], the dominant factor affecting the performance of parallelizing over particles is I/O. The solution to solve the I/O problem during runtime is to perform prefetching of data. However, works that propose prefetching incur additional costs of making predictions of which blocks to read. Leveraging the memory hierarchy similar to Camp et al. is a good strategy, provided proper considerations for vector field size and complexity are made. Apart from I/O costs, load balancing remains another factor affecting performance adversely. Previous work stealing and work requesting strategies have demonstrated good load balance with additional costs of communicating work items. These costs could potentially be restrictive in the case of workloads with a large number of particles.

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. [108] introduced an algorithm that uses a master-worker model. The processes were divided into groups, and each group had a master process. The master is responsible for maintaining the load balance between processes as it coordinates the assignment of work. The algorithm begins with statically partitioning the data. All processes load data on demand. Whenever a process needs to load data for advancing its particles, it coordinates with the master. The master decides whether it is more efficient for the process to load data or to send its particles to another process. The method proved to be more efficient in I/O and communication than the traditional parallelization approaches.

Kendall et al. [76] provided a hybrid solution which they call DStep and works like the MapReduce framework [43]. Their algorithm used groups for processes as well and has a master to coordinate work among different groups. A static round-robin partitioning strategy is used to assign data blocks to processes, similar to Peterka et al. [102]. The work of tracking particles is split among groups where the master process maintains a work queue and assigns work to processes in its group. Processors within a group can communicate particles among them. However, particles across groups can only be communicated by the master processes. The algorithm provided an efficient and scalable solution for particle tracing and has been used by other works [60, 59, 87].

Binyahib et al [17] 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 [19]. HyLiPoD is short for Hybrid Lifeline and Parallelize over Data, and the algorithm aims to choose the best strategy between the Lifeline algorithm [18] and parallelize over data for distributed particle advection given a certain workload.

Table 4. Recommendation of parallelization strategy for particle advection workloads based on features of the problem. This table appears in the survey by Binyahib [14].

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

Table 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
[133]	Intel Xeon (8x4)	32	644M	-	1M	SL, PL	-	hierarchical representation, strong scaling
	AMD Optron (2048x2)	256	644M	100	1M		-	
[37]	Intel Xeon (48x2)	32	162M	-	700	SL	-	data partitioning, strong scaling
[108]	Cray XT5 (ORNL)	512	512M	-	4k, 22K	SL	U	data loading, data partitioning, weak scaling

	(149K)	512	512M	-	10K		U	
		512	512M	-	20K		U	
[102]	PowerPC-450	16k	8B	-	128k	SL,	RD	domain decomposition, dynamic repartitioning, strong and weak scaling
	(40960x4)	32K	1.2B	32	16k	PL	RD	
[30]	Intel Xeon	-	512M	-	2.5K, 10K	SL	D,	Effects of storage hierarchy
	Dash (SDSC)	-	512M	-	2.5K, 10K		U	
		-	512M	-	2.5K, 10K			
[32]	Cray XT4 (NERSC)	128	512M	-	2.5K, 10K	SL	D,	MPI-hybrid parallelism
	9572x4	128	512M	-	2.5K, 10K		U	
		128	512M	-	1.5K, 6K			
[100]	PowerPC-450	4K	2B	-	256K	SL	RD	workload aware domain decomposition, strong and weak scaling
	(1024x4)	4K	1.2B	-	128K		RD	
[99]	PowerPC-450	1k	8M	29	186M	FTLE	U	pipelined temporal advection, caching, strong and weak scaling
	(40960x4)	1K	25M	48	65.2M		U	
		16k	345M	36	288M		U	
		16K	43.5M	50	62M		U	
[31]	Cray XT4 (NERSC)	128	512M	-	128	SS	RK	comparison of parallelization algorithms for stream surfaces
	(9572x4)	128	512M	-	361		RK	
		128	512M	-	128		RK	
[97]	AMD Magny-Cours	1K	32M	735	1M	SL,	U	work requesting load balancing, strong scaling
	(6384x24)					PL		

[39]	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			
[61]	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	
[90]	PowerPC A2 (2048x16)	1K	25M	-	32K	SS	RK	caching, performance, strong scaling
		4K	80M	-	32K		RK	
		8K	500M	-	32K		RK	
		8K	2B	-	64K		RK	
[136]	Intel Xeon (8x8)	64	3.75M	24	6250	PL	U	data prefetching, strong scaling
		64	25M	48	4096		-	
[135]	PowerPC A2 (2048x16)	8K	1B	-	128M	SL,	-	domain decomposition, using K-d trees, strong and weak scaling
		8K	3.8M	24	8M	S-D,	-	
		8K	25M	48	24M	FTLE	U	
[16]	Intel Xeon (2388x32)	512	67M	-	1M	S-D	D, U	in situ parallelization over particles
[19]	Intel Xeon (2388x32)	1K (8K cores)	34B	- -	343M	S-D	D, U	comparison of parallelization algorithms
[17]	Intel Xeon (2388x32)	1K (8K cores)	34B	- -	343M	S-D	D, U	novel hybrid parallelization algorithm

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

2.5.2.2 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 based on various workload factors discussed earlier in the section. These strategies were presented in a survey by Binyahib [14]. ***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.

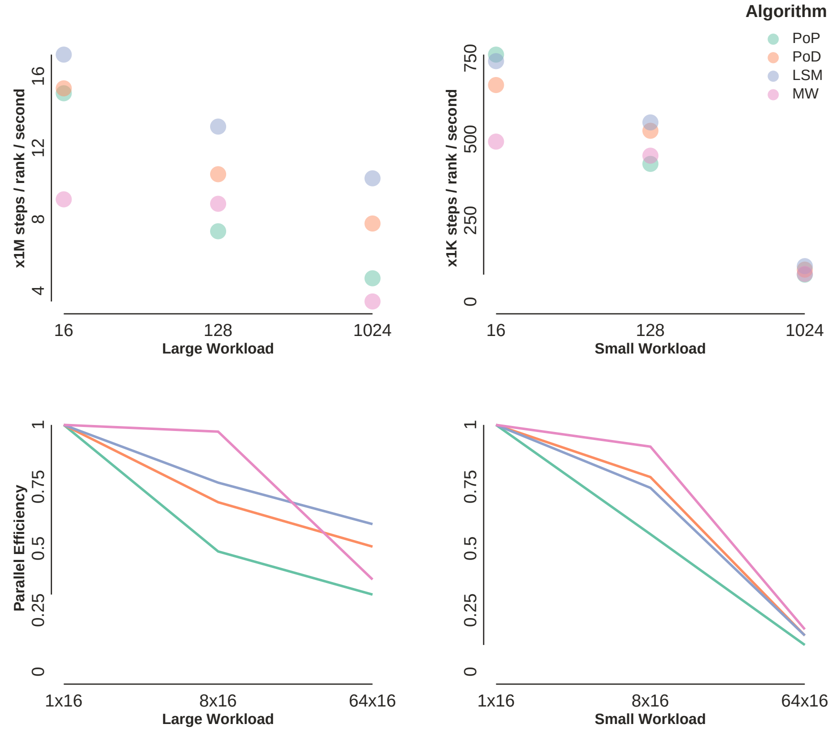


Figure 5. Weak scaling plots for distributed memory particle advection based on the comparison of four parallelization algorithms by Binyahib et al. [19]. The plots present performance comparison of the algorithms for two different workloads. The large workload used 1 particle per 100 cells where each particle advanced 10K steps. The small workload used 1 particle per 10K cells where each particle advanced 1K steps. The plots on the top present the performance of the algorithms in terms of throughput along the Y axis, while the plots on the bottom present the parallel efficiency while using weak scaling along the Y axis. In all cases the X axis represents the number of MPI ranks used to perform the experiments.

Figure 5 shows a comparison of scaling behaviors of four parallelization algorithms, extracted from the study presented by Binyahib et al. [19]. These algorithms include parallelize over particles, parallelize over data, Lifeline Scheduling Method (LSM, an extension of parallelize over particles) [18], and master-worker (a hybrid parallel algorithm). The Figure presents a weak scaling of these algorithms. The top row plots show the throughput of these algorithms in terms of number of steps completed by each MPI rank per second. The bottom row plots show the efficiency of weak scaling achieved by the different algorithms. The efficiency of the algorithms drop significantly as the concurrency and workload 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 speedups 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. [37], which used only 32 processes to produce 700 streamlines. The work with the largest number of processes was by Nouanesengsy et al. [99], which used 16 thousand processes for FTLE calculation. However, the work with the most workload was by Binyahib et al. [19], 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.

2.5.3 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.[32] presented two approaches that used multi-core processors to parallelize particle advection 1) parallelization over particles, and 2) parallelize over data blocks. In both cases, the authors aimed to use the N allocated cores. For parallelization over particles, N worker threads were used along with N I/O threads. The worker threads are responsible for performing particle advection. The I/O threads manage the cache of data blocks to support the worker threads. For parallelization over data blocks, $N-1$ worker threads are used, which access the cache of data blocks directly, and an additional thread was used for communicating results with other processes.

Camp et al. [33] 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. [39] explored particle advection performance across various GPUs (counts and device) and CPUs (processors and concurrency). Their objective was to explore the relationship between parallel device choice and the execution time for particle advection. Two of their key findings were: 1) For CPUs, adding more cores benefited workloads that execute for medium to longer duration, 2) CPUs with many cores were as performant as GPUs and often outperformed GPUs for small workloads with short execution times. 3) With higher particle densities (50^3 or more) GPUs can be saturated and result in performance imporvements proportpional to their FLOP rates, faster GPUs can provide better speedups.

Jiang et al. [73] 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 oversubscription of streamline tasks is an effective method for hiding I/O latency, which is bottleneck for particle advection.

2.6 Conclusion and Future Work

This survey has provided a guide to particle advection performance. The first two parts focused on high-level concepts: particle advection building blocks and a cost model. The last two parts surveyed existing approaches for algorithmic optimizations and parallelism. While the guide has summarized research findings to date, additional research can make the guide more complete. Looking ahead to future work, we feel this survey has illuminated three types of gaps in the area of particle advection performance.

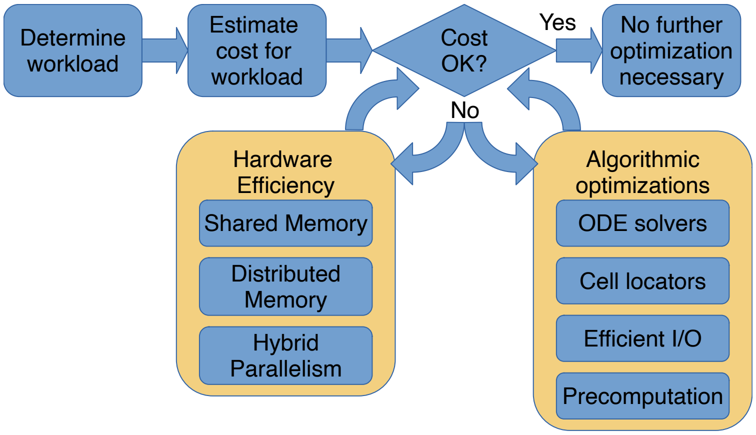


Figure 6. A flow chart to determine the potential optimizatios to be applied to a flow visualization algorithm.

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 [110], significantly more detail would be useful. To bridge this gap, the study to estimate the speedups for particle advection on diverse GPUs is discussed in Chapter IV.

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. [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. Finally, there are additional types of optimizations. Taking another example from ray tracing, Morrical et al. [96] presented a method that improved the performance of direct unstructured mesh point location [117] by using 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. While our cost model is useful in the context of providing a guide for particle advection performance, future work could make the model more powerful. One possible extension relates to the first type of gap (lack of holistic studies). In particular, holistic studies on expected speedups over diverse workloads would enable adaptive step sizing and GPU acceleration to fit within the model. Incorporating the findings for GPU acceleration from Chapter IV to test the cost model still remains unexplored. Another possible optimization is to broaden the model. In particular, I/O is not part of our cost model, so optimizations for I/O efficiency cannot be included at this time. Further, precomputation likely requires a different type of model altogether, i.e., using a form of the current model for regular particle advection and a different model for precomputation, and selecting the best approach from the two. Finally, distributed-memory parallelism is often applied to very large data sets that cannot fit into memory of a single node, which significantly increases modeling complexity. That said, these extensions could have significant benefit. One benefit 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 flow chart in Figure 6, 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.

CHAPTER III

PORTABLE PARTICLE ADVECTION USING VTK-M

The contents for this chapter come from published co-authored work [110], where David Pugmire was the first author. I was the second author of the paper. David designed and implemented the initial algorithm. This design is included in the dissertation for reference. My contributions were in leading the evaluation, as I was primarily responsible for designing the experiments and running all experiments. I also led the efforts to analyze and interpret the results. Hank Childs, Bernd Hentschel, and David Pugmire all assisted in interpreting the results and editing. David Pugmire also played a major role in writing the sections not involving experiment design and analysis.

Particle advection is the fundamental kernel behind most vector field visualization methods. Yet, the efficient parallel computation of large amounts of particle traces remains challenging. This is exacerbated by the variety of hardware trends in today's HPC arena, including increasing core counts in classical CPUs, many-core designs such as the Intel Xeon Phi, and massively parallel GPUs. The dedicated optimization of a particle advection kernel for each individual target architecture is both time-consuming and error prone. In this chapter, we propose a performance-portable algorithm for particle advection. Our algorithm is based on the recently introduced VTK-m system and chiefly relies on its device adapter abstraction. We demonstrate the general portability of our implementation across a wide variety of hardware. Finally, our evaluation shows that our hardware-agnostic algorithm has comparable performance to hardware-specific algorithms.

3.1 Introduction

In order to keep up with the amounts of raw data generated by state of the art simulations, modern visualization algorithms have to be able to efficiently leverage the same, massively parallel hardware that is used for data generation, i.e. today's largest supercomputers. This holds true for both classical *post processing* and modern *in situ* strategies. Specifically, as the latter have to run *at simulation time*, they have to be able to deal with a large variety of hardware platforms efficiently. Even more challenging, as visualization is oftentimes not seen as a first class citizen, it might have to run on different resources whenever they are available, e.g. utilizing idle CPU cores while the simulation is advanced on GPUs or using a local GPU while the simulation is blocked by communication. Custom-tailoring visualization algorithms to specific hardware platforms and potential usage scenarios is both time-consuming and error-prone. This leads to a gap with regard to practically available visualization methods for large data: either they are portable across a number of architectures, but do not feature ultimate performance, or they are highly-optimized, yet work only on a very limited subset of today's diverse hardware architectures. Performance-portable formulations of key algorithms have the potential to bridge this gap: the developer specifies *what* can be run in parallel while an underlying run-time system decides the *how* and *where*. Further, the runtime system is optimized *once* to make good use of a specific target architecture. Ideally, the result is that all previously formulated kernels will be available — with good, if not fully optimal, performance — on the newly addressed system. In the past, this approach has been demonstrated to show good results for inherently data parallel visualization problems, e.g. ray tracing [85] and direct volume rendering [84].

In this chapter, we extend the body of performance-portable approaches with a method for parallel particle advection. Particle advection is the basic algorithmic kernel of many vector field visualization techniques. Applications encompass, e.g., the direct representation of field lines [119, 47], dense vector field visualization methods [29, 127, 81], flow surfaces [48, 55, 72], and the computation of derived data fields or representations [5, 53, 65, 113] or statistical measures [131, 130]. These techniques depend on the ability to compute large numbers of particle trajectories through a vector field. The resulting workloads are taxing with respect to both their computational requirements and their inherent dependence on high data bandwidth. In contrast to the aforementioned visualization kernels — isosurfacing and ray casting — particle tracing computations are not trivially data parallel. Worse, workloads are highly dynamic, as the outcome inherently depends on the input vector field. The computations' overall demands are therefore hard to predict in advance. This complicates — among other things — the efficient scheduling of parallel particle advection computations. In the recent past, a variety parallelization strategies has been proposed, all sharing the goal of providing many particles fast (see, e.g., [27, 108, 32, 33, 38, 100, 99, 97]).

Our approach is based on the *parallelize-over-seeds* (POS) strategy. In order to facilitate performance portability, we rely on the concepts of the recently introduced VTK-m framework [95]. This facilitates the direct use of classical CPUs and GPUs without the need to implement and maintain two distinct code paths. We evaluate our algorithm's performance for a set of five well-defined workloads which cover a wide range of vector field visualization tasks. Our results show that our technique performs as expected, and matches the performance characteristics identified in previously published work. Further, we demonstrate that there are minimal performance penalties in our portable implementation when compared to hand-coded reference implementations.

To summarize, we make the following contributions. We propose a performance-portable formulation of POS particle tracing embedded in the VTK-m framework. We demonstrate general effectiveness and performance-portability based on the results of several performance experiments. As part of these experiments, we assess the cost of performance portability by comparing our method's performance to that of native implementations on a variety of execution platforms. Against this backdrop, we discuss the advantages and limitations of our approach with respect to natively-optimized techniques.

3.2 Related Work

In this section, we briefly review related work under two different aspects: parallel visualization systems and particle advection.

Parallel Visualization Systems The increasing need for production-ready, scalable visualization methods led to the development of several general purpose packages such as Paraview [6, 9] and VisIt [40]. These tools have primarily focused on distributed-memory parallelism, which is complementary to our own focus.

Recent changes in both HPC hardware and software environments led to the development of new frameworks, which address certain aspects of the changing HPC environment.

The DAX toolkit [93], introduced by Moreland et al., is built around the notion of a *worklet*: a small, stateless construct which operates — in serial — on a small piece of data. Worklets are run in an execution environment under the control of an executive. They help programmers to exhibit fine-grained data parallelism, which is subsequently used for data-parallel execution.

Lo et al. proposed to use data parallel primitives (DPP) [21] for a performance-portable formulation of visualization kernels [88] in the Piston framework. Based on NVidia's Thrust library [11], Piston supports both GPUs and CPUs as target architectures, by providing a CUDA and an OpenMP backend, respectively. Performance results demonstrated the ability to achieve good performance on different platforms using the exact same source code for each.

Meredith et al. introduce EAVL, a data parallel execution library with a flexible data model that addresses a wide range of potential data representations [92]. With this data model, they aim at increased efficiency – both in terms of memory use and computational demand – and scalability. The generic model proposed for EAVL allows developers to represent (almost) arbitrary input data in a way that accounts for hardware-specific preferences. For example, it allows them to switch between *structure-of-arrays* and *array-of-structures* representations of multi-dimensional data fields. Low-level parallelism is supported by an iterator-functor model: iteration happens in parallel, applying a functor to each item of a range.

Eventually, the experiences gathered in the development of these libraries resulted in the consolidated development of VTK-m [95]. It integrates an evolution of EAVL's data model with the two-tier control/execution environment of DAX and the idea to facilitate performance portability by formulating visualization workloads in terms of data parallel primitives. Currently, it offers parallel backends for CUDA and Intel Threading Building Blocks (TBB). Moreland et al. discuss the cost of portability for a variety of visualization workloads. Their findings are inline with and partially based on work by Larsen et al., who studied DPP-based formulations for ray tracing and direct volume rendering, respectively [85, 84]. In this chapter, we focus on an efficient formulation of a basic, general-purpose particle advection kernel running in a shared memory parallel environment. Hence, we chose VTK-m as our development platform.

Particle Advection The computation of integral lines is a fundamental kernel of vector field visualization [91]. Pugmire et al. review the two fundamental approaches – *parallelize-over-seeds* (POS) and *parallelize-over-blocks* POB – in a distributed memory environment and introduce a hybrid master-slave scheme that addresses load-balancing issues [108]. POS distributes the seeds of a target particle population across processing elements (PEs) and computes them independently of each other. In contrast, POB assigns the individual blocks of a domain decomposition to PEs; then, each PE is responsible for generating the traces that enter one of its assigned blocks. The newly introduced hybrid scheme, which dynamically – and potentially redundantly – assigns blocks to PEs addresses load-balancing problems that typically become a problem for pure POB while also limiting redundant I/O operations which oftentimes limit POS' scaling. An overview of distributed memory parallel particle advection methods is given in [109].

Camp et al. propose a two-tier scheduling scheme: they use MPI to parallelize computations across multiple nodes and then execute local advection using OpenMP-parallel loop constructs [32]. Subsequently, Camp et al. analyze the effects exchanging the OpenMP-based advection for a CUDA-based solution [33]. We use a refined version of Camp's original CUDA advection scheme for comparisons in Sec. 3.4.

Kendall et al. propose a method inspired by MapReduce to parallize the domain travseral inherent to parallel particle tracing [76].

Yu et al. propose a data-dependent clustering of vector fields which enables the development of a data-aware POB strategy [133]. Nouanesengsy et al. accumulate information about the probable propagation of particles across blocks and subsequently formulate the task of partitioning the set of blocks to PEs as an optimization problem [100]. Subsequently, Nouanesengsy et al. propose to extend the basic POB idea to the time dimensions, distributing time intervals to PEs in order to generate the pathlines needed to compute the Finite Time Lyapunov Exponent (FTLE) [99].

Guo et al. propose a method using K-D tree decompositions to dynamically balance the workload for both steady and unsteady state particle advection [135].

Mueller et al. investigate the use of an alternative, decentralized scheduling scheme, *work requesting*: whenever a process runs out of work – i.e. particles to integrate – it randomly picks a peer and requests half its work in order to continue [97]. Hence scheduling overhead only occurs when there is actual imbalance in the system.

Being a key computational kernel for an array of vector field visualization algorithms, the optimization of particle advection for different architectures has garnered significant interest in the visualization research community. Initially, this was fueled by the advent of modern, programmable graphics hardware (GPUs). Interactive particle integration has been targeted for steady-state uniform grids [27], time-varying uniform grids [27], and tetrahedral meshes [118], respectively. Bussler et al. propose a CUDA formulation for particle advection on unstructured meshes and additionally investigate the use of 3D mesh decimation algorithms in order to reduce the GPU memory requirements. Hentschel et al. propose the tracing of particle packets which facilitates the use of SIMD extensions in modern CPU designs [68]. They found that significant gains can be achieved specifically due to the optimization of memory accesses. Chen et al. follow a similar idea: they propose to integrate spatially coherent bundles of particles through time-varying, unstructured meshes in order increase memory locality on the GPU [38].

The studies presented in [33, 39] compare CPU and GPU-based hardware architectures w.r.t. their suitability for parallel integral curve computations. Sisneros et al. performed a parameter space study for a round-based POB advection algorithm [122]. Their findings suggest that naïvely chosen default settings – e.g. advecting all particles in each round – often lead to significantly degraded performance.

In summary, we find a great variety of optimization efforts targeting the important yet seemingly inconspicuous computational kernel of particle integration. In light of studies like [33, 39, 68], we argue that making good use of any new hardware architecture or even of new features on the one hand requires an intimate knowledge of said features and on the other hand can be very time consuming, particularly due to the required low-level programming. This observation provides the major motivation for the performance-portable approach proposed in the following section.

3.3 Parallel Particle Advection in VTK-m

As stated above, the VTK-m framework is a response to the growing on-node parallelism that is available on a wide variety of new architectures. In order to be able to efficiently cater to a variety of different hardware platforms, VTK-m relies on the concept of data parallel primitives. VTK-m distinguishes two different realms: the control environment and the execution environment. The control environment is the application-facing side of VTK-m. It contains the data model and allows application programmers to interface with algorithms at large. In contrast, the execution environment contains the computational portion of VTK-m. It is designed for massive parallelism. Worklets (c.f. Sec. 3.2) are an essential part of the execution environment, and are the mechanism for performing operations on elements of the data model in parallel. Finally, device adapters provide platform-specific implementations of generic DPPs and memory management. Specifically — where necessary — they abstract the transfer of data between host and device memory.

Algorithms in VTK-m are created by specifying a sequence of DPP operations on data representations. When compiled, the parallel primitives are mapped to the particular device implementation for each primitive. This indirection limits the amount of optimization that can be done for a particular algorithm on a particular device, which in turn raises the question of costs for performance portability. In the following, we describe a DPP-based realization of parallel particle advection which will eventually lead to a parallelize-over-seeds scheme.

3.3.1 A Data-Parallel Formulation of Particle Advection.

In formulating a design for particle tracing using the DPP in VTK-m, our primary task was to determine the definition for the elementary unit of work. This elementary work unit can then be mapped onto the set of execution threads to perform the total amount of work using massive parallelism. The most natural elemental unit of work is the advection of a single particle. However there are subtleties in providing a precise definition. The traditional option, and the one we selected, is to define the unit of work as the entire advection of an individual

```
INPUTS:
Seed, VectorField, NumberOfSteps, StepSize
OUTPUTS:
Advected

Advected = Advect(Seed, VectorField,
                  NumberOfSteps, StepSize)

Function:
Advect(pos, vectorField, numberOfSteps, stepSize):
  S = 0
  while S < numberOfSteps :
  {
    if pos in vectorField :
      newPos = RK4(pos, stepSize, vectorField)
      pos = newPos
      S = S+1
    else :
      break
  }

  return pos

Function:
RK4(p, h, f)
   $k_1 = f(p)$
   $k_2 = f(\frac{h}{2}k_1)$
   $k_3 = f(p + \frac{h}{2}k_2)$
   $k_4 = f(p + hk_3)$
   return $\frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4)$
```

Figure 7. Pseudocode for our implementation of `Advect`, our particle advection for a routine parallelize-over-seed (POS) strategy. This routine operates on a single particle and provides the definition of our elementary unit of work.

particle. Other options include those studied in [122] where the unit of work is defined as a fixed number of advection steps for an individual particle, or a group of particles. These other options provide smaller granularity which provide opportunities for better load balancing of total work. This, however, comes at the cost of increased overhead for scheduling.

For our study, we decided to define the entire advection of a single particle as the elementary unit of work for the following reasons. First, this choice naturally encodes a map from an input position — the seed location — to an output position — the advected particle. Second, this map can directly be expressed by a data parallel primitive. Finally, since this is the most widely used approach, we felt it was imperative to thoroughly understand this method as it would inform directions for future improvements.

The pseudocode in Figure 7 shows the implementation of our elementary unit of work, the `Advect(...)` function. The input to this function consists of a seed location, a vector field, an integration step size, and the maximum number of integration steps. The particle trajectory is computed using a numerical integration scheme. For this chapter, we use the well-established 4^{th} order Runge-Kutta method. Advection terminates when a maximum number of steps is reached, or when the particle leaves the spatial domain of the vector field. To advect multiple seeds, the `Advect` kernel is applied to each individual seed.

The advection of each seed position, i.e. each loop iteration, is independent of all other seed locations. This leads to the following two observations: First, the computation for each seed can be performed in parallel without the need for any form of synchronization. Second, as stated above, this forms a basic unit of work that is to be executed per seed. Hence, we express the handling of a single seed

```
INPUTS:
Seeds, vectorField, numberOfSteps, stepSize
OUTPUTS:
Advected

 Advected = map<Advect>(Seeds,
                        vectorField,
                        numberOfSteps,
                        stepSize)
```

Figure 8. Pseudocode for our implementation of particle advection using DPP. The Advect function is defined in Figure 7. It serves as the functor to the map DPP which calls it – in parallel – for all seeds. The DPP is shown in the form of primitive<functor>(arguments).

by means of a functor that operates on an elementary piece of data — the seed location and wrap this functor as a VTK-m worklet. In order to keep the worklet description hardware-independent, it is important to note that all accesses to raw memory are encapsulated by so called array handles. In this way, the exact location of a data item in memory — specifically if it resides in host or device memory — is hidden from the worklet. This enables the flexible, automatic management of the actual memory by the underlying device adapter.

With the elementary operation of advection of an individual particle formulated in a generic fashion, the remaining task is to enable a concurrent execution for multiple particles. This is realized by means of a specific data parallel primitive: the `map` operation. In this specific case, we aim to *map* an input seed position to its eventual end position after advection. The map operation is one of the DPPs which is implemented in a highly optimized form by the VTK-m runtime. Hence its use — illustrated in Figure 8 — entails a platform-specific, parallel execution of the `Advect` functor on each particle on the underlying parallel hardware. In particular, if the runtime environment is an accelerator device, all data that is consumed or produced by the advection operation is automatically transferred to/from device memory. In this way, the decision of where the advection operation is executed is completely hidden from the developer of the worklet. This is the main reason why worklets can access data only via so called array handles (see above). Using the *map* DPP, we now have obtained a data parallel formulation of particle advection that resembles the basic parallelize-over-seeds principle.

Finally, we note that for the purposes of this study we are focused exclusively on the performance of particle advection techniques where only the final location of the seeds is computed. This is the exact formulation for analysis techniques such as FTLE, and a fundamental building block for other techniques which use the saved trajectories of seeds such as streamlines, pathlines, streamsurfaces, and Poincaré methods.

3.4 Performance Evaluation

In this section we discuss the experimental setup, including data sets, workloads, and hardware platforms, followed by the performance of our approach on each experiment.

3.4.1 Experimental Setup. In order to evaluate our work, we have selected a number of different experiments that cover a range of uses cases for particle advection. We use three different parameters for our study, which we vary independently. The first parameter is the vector field data. The types of flow structures present in a data set has a tremendous impact on the performance of an implementation, and so we capture various types of flows by using multiple data sets. The second parameter is the workload, which consists of the number of particles and the number of integration steps. The various types of vector field analysis techniques tend to use different classes of workloads, and so we have

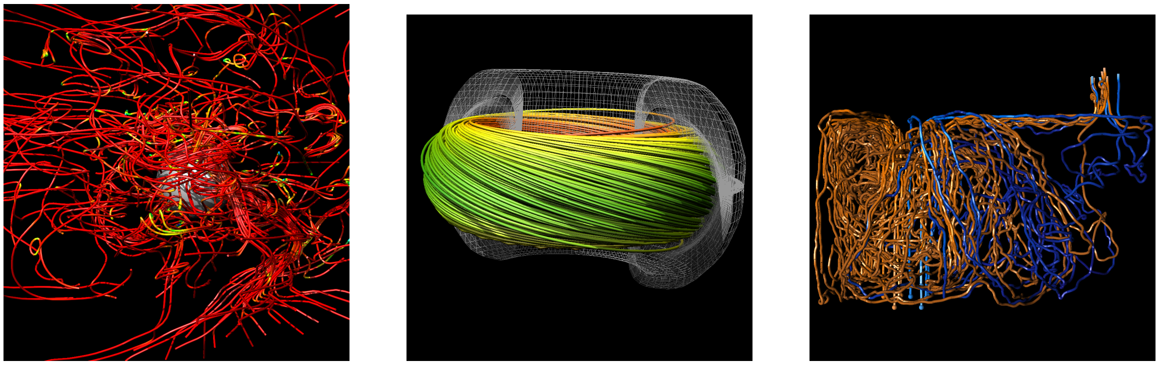

(a) Astro (b) Fusion (c) Thermal Hydraulics

Figure 9. Sample streamlines show typical vector field patterns in the three data sets used in this study.

selected a set of workloads that capture the common use cases. The third and final parameter that we vary in our study is the execution hardware in order to capture performance on both CPU and GPU environments. We vary this third parameter at an even finer level of granularity by performing each test on several different types of CPUs and GPUs.

Data Figure 9 provides information on the three different data sets used in our study. The *Astro* data set contains the magnetic field surrounding a solar core collapse resulting from a supernova. This data set was generated by the GenASiS [46] code which is used to model the mechanisms operating during these solar core collapse events. The *Fusion* data set contains the magnetic field in a plasma within a fusion tokamak device. This data set was generated by the NIMROD [123] simulation code which is used to model the behavior of burning plasma. The plasma is driven in large measure by the magnetic field within the device. Finally, the *Thermal Hydraulics* data set contains the fluid flow field inside a chamber when water of different temperatures is injected through a small inlet. This data set was generated by the NEK5000 [49] code which is used for the simulation computational fluid dynamics. The vector fields for all of our representative data sets are defined on uniform grids. All three data sets feature a resolution of $512 \times 512 \times 512$ accounting for $1,536$MB per vector field. We have specifically chosen the simplest type of data representation to exclude additional performance complexities that can manifest with more complex grid types, e.g., point location in unstructured meshes.

Workloads As stated above, a workload consists of a set of particles and the number integration steps to be taken. The particles are randomly distributed throughout the spatial extents of the data set grid. We chose a set of five workloads that we felt mimicked the behavior of common uses cases, e.g., from streamlines to FTLE computations. Further, we specifically choose this set based on work by Camp et al. [33], who studied the performance of both CPU and GPU implementations, and identified workloads that were well suited to each architecture.

These five workloads are defined as follows:

- $\mathbf{W_1}$: 100 seeds integrated for 10 steps.
- $\mathbf{W_2}$: 100 seeds integrated for 2000 steps.
- $\mathbf{W_3}$: 10M seeds integrated for 10 steps.
- $\mathbf{W_4}$: 10M seeds integrated for 100 steps.
- $\mathbf{W_5}$: 10M seeds integrated for 1000 steps.

Table 7. Hardware used in performance portable particle advection study.

Machine	**CPU**	**GPU**
Rhea Partion 1	Dual Intel Xeon E5-2650 "Ivy Bridge", 2.0 GHz 16 total cores 128 GB RAM	*None*
Rhea Partition 2	Dual Intel Xeon E5-2695 v3 "Haswell", 2.3 GHz 28 total cores 1 TB RAM	2x NVIDIA K80 12 GB Memory
Titan	*Not used*	NVIDIA K20X 6 GB Memory
SummitDev	Dual IBM Power8 3.5 GHz 20 total cores 256 GB RAM	4x NVIDIA P100 16 GB Memory

Hardware The execution environment for our study consists of the three systems deployed at the Oak Ridge Leadership Compute Facility (OLCF) (c.f. Table 7).

- Titan is a Cray XK7, and is the current production supercomputer in use at the OLCF. It contains 18,688 compute nodes and has a peak performance of 27 petaflops.
- Rhea is a production cluster used for analysis and visualization via pre- or post-processing. It is a 512 node commodity Linux cluster that is configured in two partitions. The first partition is targeted for processing tasks requiring larger amounts of memory and/or GPUs. The nodes in its second partition do not have GPUs.
- SummitDev is an early access 54 node system that is one generation removed from Summit, the next supercomputer that will be installed at the OLCF.

We note that both Rhea and SummitDev contain multiple GPUs on each node, however in this study we are only studying our implementation on a single GPU.

Compilers Our VTK-m code was compiled on Rhea and Titan using the 4.8.2 version of the GCC compiler, the most stable version for Titan. For SummitDev we used the closest available version, which was 4.8.5. On Rhea and Titan we used version 7.5.18 of CUDA as it is the most stable version for Titan. On SummitDev the only version of CUDA available was version 8.0.54. On all platforms, our code was compiled using full optimization flags, `-O3`

3.4.2 Results. In this section we present the results from our experiments. In Section 3.4.2.1 we present results for our VTK-m implementation across the workloads described above and discuss the performance. We also demonstrate the parallel efficiency of our implementation for CPUs. Finally, in Section 3.4.2.2 we compare our implementation with two hand-coded hardware specific implementations and discuss the performances.

3.4.2.1 VTK-m Results. The data in Table 8 contains the runtimes for our VTK-m implementation for the cross-product of the five workloads, the three data sets, and hardware types. We ran the GPU experiments under two different scenarios, which are shown in the first two sets of three columns each. For the first scenario, we assume that the vector field data resides in host memory and has to be uploaded to the device before tracing. The timings for this scenario are shown in the set of three columns of the table labeled “GPUs with data transfer”.

The second scenario assumes an in situ setting: the vector field resides on the device already, either because it has been generated there or because it is uploaded once for subsequent interactive exploration. This second scenario is

Table 8. Timings (in seconds) for VTK-m implementations for each test in our experimental setup.

		GPU with data transfer			GPU without data transfer			CPU		
	File	K20X	K80	P100	K20X	K80	P100	Intel$_{16}$	Intel$_{28}$	IBM P8$_{20}$
$\mathbf{W_1}$	Astro	0.627s	0.521s	0.389s	0.000s	0.011s	0.014s	0.001s	0.001s	0.001s
	Fusion	0.627s	0.521s	0.387s	0.001s	0.011s	0.015s	0.001s	0.001s	0.001s
	Thermal	0.627s	0.521s	0.392s	0.001s	0.011s	0.024s	0.001s	0.001s	0.001s
$\mathbf{W_2}$	Astro	0.648s	0.543s	0.404s	0.021s	0.033s	0.029s	0.071s	0.046s	0.053s
	Fusion	0.649s	0.543s	0.400s	0.023s	0.033s	0.028s	0.071s	0.051s	0.052s
	Thermal	0.648s	0.541s	0.395s	0.021s	0.031s	0.027s	0.074s	0.048s	0.051s
$\mathbf{W_3}$	Astro	1.511s	0.946s	0.577s	0.884s	0.436s	0.202s	3.003s	1.257s	2.327s
	Fusion	1.509s	0.961s	0.582s	0.883s	0.451s	0.210s	2.948s	1.208s	2.609s
	Thermal	1.508s	0.945s	0.583s	0.881s	0.435s	0.215s	2.801s	1.179s	2.691s
$\mathbf{W_4}$	Astro	5.193s	2.851s	1.765s	4.566s	2.341s	1.390s	28.702s	10.688s	20.708s
	Fusion	5.327s	2.795s	1.776s	4.701s	2.285s	1.404s	26.295s	10.785s	19.949s
	Thermal	5.099s	2.785s	1.777s	4.472s	2.275s	1.409s	26.641s	11.266s	19.365s
$\mathbf{W_5}$	Astro	38.660s	23.322s	13.338s	38.033s	22.812s	12.963s	256.900s	107.806s	185.852s
	Fusion	41.116s	24.450s	13.648s	40.490s	23.940s	13.276s	272.165s	107.113s	186.455s
	Thermal	39.444s	24.153s	13.626s	38.817s	23.643s	13.258s	260.740s	106.881s	193.110s

representative for, e.g., an in situ change of the data's representation [5] or an exploratory visualization where particles are seeded and displayed in a highly interactive fashion [27, 118]. The timings for this in situ scenario are shown in the set of three columns of the table labeled "GPUs without data transfer".

For workloads with few particles (e.g., $\mathbf{W_1}$ and $\mathbf{W_2}$) the overhead for data transfers to the GPU is clearly evident. As would be expected, as more work is available for the GPUs this data transfer overhead can be better amortized over the particle advection work.

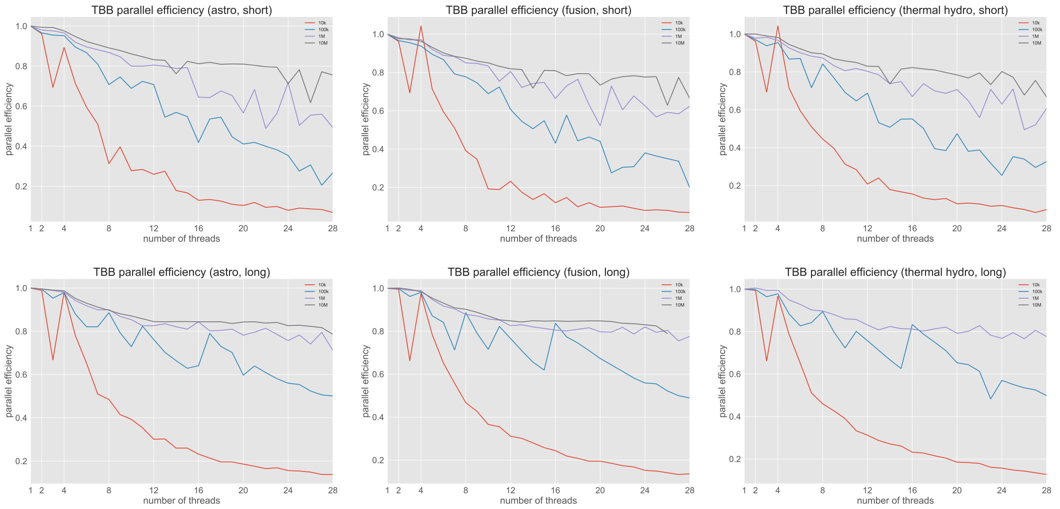
Analogous timings for various CPU settings are shown in the last three columns of Table 8.

In comparing the GPU and CPU implementations, we offer the following observations. CPUs tend to perform better than GPUs for lower seed counts. We observe this behavior in workloads $\mathbf{W_1}$ and $\mathbf{W_2}$ where a small number of seeds are advected very short and very long distances respectively. In contrast, GPUs tend to perform better with higher seed counts, and more advection steps. We observe this behavior in workloads $\mathbf{W_3}$, $\mathbf{W_4}$, and $\mathbf{W_5}$. In workload $\mathbf{W_3}$, which features a large number of particles advected for a medium number of steps, we see only moderate wins for the GPU. These observations are in line with earlier studies by Camp et al. [32, 33].

We note that the particle advection source code was identical for all tests; in particular, it was not hand-tuned to any hardware platform. Hence, these results provide evidence of the portable performance of a single implementation on a broad set of execution environments.

In addition, we see the expected trends in performance across hardware families. For example, the NVIDIA P100 is faster than the K80, which in turn outperforms the K20X. We see a similar trend when comparing the times for the two different generations of Intel processors on each of the Rhea partitions. We also note that the performance on the IBM Power8 CPUs that contain 20 cores each falls between the performance of the 16 and 28 core Intel processors, which aligns with expectations.

Finally, we are interested in understanding the scalability of the VTK-m implementation on the CPU. Figure 10 shows the efficiency with respect to number of cores used for several workloads run on the Rhea Partition 2 CPU. For low seed counts, parallel efficiency is lacking. Quite simply, there is not enough concurrently executable work in the system to enable efficient scheduling and thus good resource utilization. However, once enough parallelism becomes available – afforded by increased number of particles – we see excellent efficiency: for short duration workloads, and populations of 1 million particles or more, parallel efficiency remains above 60%; for longer duration workloads, the efficiency is around 80%.


TBB parallel efficiency (astro, short)

TBB parallel efficiency (fusion, short)

TBB parallel efficiency (thermal hydro, short)

TBB parallel efficiency (astro, long)

TBB parallel efficiency (fusion, long)

TBB parallel efficiency (thermal hydro, long)

Figure 10. Parallel efficiency for the VTK-m implementation running on the CPU of the Rhea Partition 2. The workloads shown are for $10k$, $100k$, $1M$, and $10M$ seeds on all three data sets. The top row shows advections for short durations, and the bottom row shows advections for long durations. Note: Our allocation was exahusted before we could complete the data collection for the $10M$ seed case in the bottom, far right, and so these numbers are not included. Overall, we feel the scaling behavior is good, as efficiency often drops as more and more cores are used.

3.4.2.2 Comparisons to Other Implementations. Since our implementation is per definition agnostic of the eventual execution environment, we are particularly interested in any performance penalties due to portability. To explore these impacts, we compare our results to two different reference implementations. These comparison codes were run on the same hardware, and compiled using the same compilers as our VTK-m implementation.

First, we compare our code to hand-coded implementations for CPUs and GPUs using pthreads and CUDA, respectively. The reference code has been evaluated in [32, 33].

The data in Table 9 lists the runtimes for each workload using the CUDA specific implementation, and a performance factor for the VTK-m runtimes.

This factor gives the relative speedup of our implementation over the reference implementation, i.e. factors larger than 1 indicate our implementation is faster. For many of the tests run, we observe that the VTK-m version outperforms the hand-coded implementation by factors up to $4X$.

These performance improvements are largely a function of two differences. The first difference is that VTK-m's CUDA device adapter performs all global device memory accesses through texture cache lookups. Hence, it is able to perform random accesses at a granularity of 32 bytes per load instead of 128 bytes. This gives the VTK-m implementation a significant advantage for workloads that heavily depend on highly random read operations, such as particle advection. Second, we note that the reference implementation is a more fully-featured system that can be run in a distributed memory parallel setting using several parallelization strategies (e.g., POS and POB). As such there are overheads associated with running this code on a single node using a POS approach. A combination of these factors explains the good performance of the VTK-m implementation.

We also note tests where the hand-coded implementation outperforms the VTK-m version.

First, for the Astro data set in $\mathbf{W_5}$, the hand-coded implementation performs significantly better. In the particular vector field for this data set, there are regions of the flow where particles quickly exit the grid. Rapidly terminating particles, however, induce imbalanced workloads which in turn is detrimental to overall performance. The hand-coded implementation handles these situations better than our implementation, and as a result achieves better performance for high workloads. We see this same situation in $\mathbf{W_4}$ for the Astro data set, but since this workload consists of less work than that of $\mathbf{W_5}$ the impact of this imbalance is not nearly as dramatic.

Second, we note performance on the K80 GPU for workloads $\mathbf{W_1}$ and $\mathbf{W_2}$. For these tests the performance of the VTK-m implementation is roughly half. We suspect that the hand-coded CUDA implementation has better work management for the low seed counts of $\mathbf{W_1}$ and $\mathbf{W_2}$ on the K80 architecture, but we are at a loss to provide a specific explanation. We note that for the performance for workloads $\mathbf{W_1}, \mathbf{W_2}$ on the K80 GPU is better than the performance on the newer P100 GPU. For workload $\mathbf{W_3}$, where there are more particles, the difference in performance between the K80 and P100 is less dramatic. For workloads $\mathbf{W_4}$ and $\mathbf{W_5}$ where there is much more work, the performance maps directly onto the generation of the GPU, as expected. We note that the hand-coded GPU implementation we are using to compare was written, tuned and optimized in the time frame of the Kepler generation of GPUs. It is possible that such optimizations specific to a particular generation of hardware might not perform well on later generation hardware and need to be optimized differently.

The data in Table 10 lists the runtimes for each workload using the pthreads specific code path of the reference implementation, and a performance factor for the VTK-m runtimes. For workloads where there is less work to be done ($\mathbf{W_1}$, $\mathbf{W_2}$), the hand-coded phtreads code performs much better than the VTK-m implementation. However, it should be noted that when there are very few particles, run times are very small, and the overheads associated with each implementation tend to dominate the comparison.

In subsequent workloads, where is more work to be performed, the performance of our VTK-m implementation is comparable in many instances.

Table 9. Timings (in seconds) for the GPU comparison implementation along with a performance factor for the VTK-m timings (factors > 1X indicate VTK-m is faster).

		CUDA Code			VTK-m Comparison		
	File	K20X	K80	P100	K20X	K80	P100
$\mathbf{W_1}$	Astro	0.844s	0.285s	0.836s	1.35X	0.55X	2.15X
	Fusion	0.845s	0.284s	0.838s	1.35X	0.55X	2.17X
	Thermal	0.844s	0.284s	0.837s	1.35X	0.54X	2.14X
$\mathbf{W_2}$	Astro	0.869s	0.301s	0.842s	1.34X	0.55X	2.08X
	Fusion	0.874s	0.304s	0.845s	1.35X	0.56X	2.11X
	Thermal	0.871s	0.304s	0.844s	1.34X	0.56X	2.14X
$\mathbf{W_3}$	Astro	3.418s	1.959s	2.353s	2.26X	2.07X	4.08X
	Fusion	3.367s	1.824s	2.219s	2.23X	1.90X	3.81X
	Thermal	3.327s	1.856s	2.247s	2.21X	1.96X	3.85X
$\mathbf{W_4}$	Astro	6.682s	5.067s	3.564s	1.29X	1.78X	2.02X
	Fusion	8.803s	6.763s	4.420s	1.65X	2.42X	2.49X
	Thermal	8.793s	6.830s	4.500s	1.72X	2.45X	2.53X
$\mathbf{W_5}$	Astro	14.353s	12.963s	6.464s	0.37X	0.56X	0.48X
	Fusion	63.993s	54.670s	25.694s	1.56X	2.24X	1.88X
	Thermal	56.133s	49.172s	23.161s	1.42X	2.04X	1.70X

We note that in general, the VTK-m implementation performs better on the Intel CPUs. This fact is not too surprising given that TBB, an Intel product, is likely optimized for Intel hardware.

One outlier in the Table 10 that is worth exploring is $\mathbf{W_5}$ for the Astro and Thermal data sets. As discussed above for the GPU implementations, the quickly exiting particles in the Astro data set leads to imbalance. We are seeing this same effect in the CPU implementation. For the Thermal data set, there is a similar issue. In the Thermal data set there are regions of the flow where particles stagnate due to zero velocities. These stagnating particles can also lead to load imbalance which in turn is detrimental to overall performance. The GPUs have enough parallelization to better amortize these stagnant particles, but in CPUs where there is less parallelization, these effects cannot be overcome. The hand-coded pthreads

Table 10. Timings (in seconds) for the CPU comparison implementation along with a comparison factor to the VTK-m timings (factors > 1X indicate VTK-m is faster).

		pthreads Code		VTK-m Comparison	
	File	Intel_{28}	IBM P8_{20}	Intel_{28}	IBM P8_{20}
$\mathbf{W_1}$	Astro	0.0006s	0.0002s	0.59X	0.17X
	Fusion	0.0004s	0.0001s	0.43X	0.09X
	Thermal	0.0004s	0.0001s	0.43X	0.07X
$\mathbf{W_2}$	Astro	0.001s	0.003s	0.03X	0.05X
	Fusion	0.003s	0.012s	0.05X	0.22X
	Thermal	0.001s	0.006s	0.03X	0.12X
$\mathbf{W_3}$	Astro	2.001s	2.408s	1.59X	1.03X
	Fusion	1.389s	2.137s	1.15X	0.82X
	Thermal	1.048s	1.719s	0.89X	0.64X
$\mathbf{W_4}$	Astro	11.675s	14.227s	1.09X	0.69X
	Fusion	11.247s	18.076s	1.04X	0.91X
	Thermal	8.123s	14.633s	0.72X	0.76X
$\mathbf{W_5}$	Astro	38.693s	53.015s	0.36X	0.29X
	Fusion	84.129s	156.735s	0.79X	0.84X
	Thermal	54.591s	113.881s	0.51X	0.59X

code handles these situations better than our VTK-m implementation, and as a result, achieves better load balancing in these situations. The issues identified in both of these data sets are planned on being addressed in the future for our VTK-m implementation.

Our final comparison is made with a fully featured production visualization and analysis software tool. The data in Table 11 contains a comparison of our VTK-m implementation to VisIt [40]. These tests were run on a CPU on the Rhea Partition 1, and compiled with the same compiler and options as our VTK-m implementation.

VisIt uses a serial execution model, and so we compare two different workloads on all three data sets using a single core execution of the VTK-m

Table 11. Timings (in seconds) for the VisIt implementation on two different workloads along with a comparison factor to the VTK-m timings (factors > 1X indicate VTK-m is faster).

			VTK-m	
	File	VisIt	1 core	28 core
100 Seeds 1000 Steps	Astro	0.0543s	2.36X	2.36X
	Fusion	0.0855s	3.56X	3.56X
	Thermal	0.0628s	2.61X	2.61X
10,000 Seeds 1000 Steps	Astro	5.5253s	2.46X	8.55X
	Fusion	7.9353s	3.22X	12.53X
	Thermal	5.9484s	2.45X	9.64X

implementation. We also provide timings made with a 28 core execution for the VTK-m implementation for additional comparisons.

On the single core example, we see a clear performance increase of VTK-m that is $2 - 3X$ faster than the VisIt implementation across both workloads. For the 28 core example, there is not enough work in the small workload to see improvements in performance when more cores are used.

However, for the larger workload we see increased performance when using 28 cores, as is expected. We note, that VisIt is a fully featured production tool, and so there are overheads associated with the implementation in VisIt. Further, the tests run in VisIt are computing streamlines, as opposed to simply advecting seed locations. As such, there are overheads associated with the storing and managing of the particle trajectories.

3.4.3 Discussion. Overall, we argue that our results support our aim of creating a performance-portable formulation for particle tracing. Across all workloads, our implementation usually outperforms the reference codes. In rare cases it takes around twice the time to complete a given benchmark, with the worst case scenario ($\mathbf{W_5}$, Astro on Power8) taking approximately 3.5× as long as the reference implementation. We note that for most of these cases we understand the reasons for the performance, and are planning to address these cases in the future.

Moreover, our implementation — by virtue of the underlying VTK-m runtime — shows good scaling on multi-core, shared memory machines. However, during preliminary experiments we discovered an aspect that affected both the TBB and the CUDA backends of VTK-m. Specifically, we noticed that the performance behavior of our test workloads was susceptible to changes in the underlying runtime's granularity settings: for TBB this would be the `grainsize` whereas for CUDA it would be the `blocksize` parameter for 1D scheduling. Changing both had a significant effect on performance. For the TBB device adapter, our experiments helped inform the decision in favor of a new, smaller default setting in VTK-m. In contrast, the general performance impact of the CUDA `blocksize` parameter on workloads outside particle advection is harder to assess and may require solutions like autotuning.

Beyond such technical issues, we observe that the performance of our implementation across the five chosen workloads matches the findings in published results that studied performance of hand coded CPU and GPU implementations. We therefore conclude that it is mostly bounded by the same limitations as the reference code. This gives us confidence that the portability of our VTK-m implementation is not introducing significant overhead issues. Further, we restate that our implementation does not contain the platform-specific optimizations that are typically found in hand-coded, hardware-specific implementations.

In summary, our findings suggest that our implementation shows competitive performance across a variety of hardware architectures and workloads.

It therefore provides a solid, and efficient basis for more sophisticated, advection-based visualization methods.

3.5 Conclusion & Future Work

In this chapter, we have introduced a performance-portable, general purpose formulation for one of the fundamental techniques for the analysis and visualization of vector fields: particle advection. Our implementation is based on the VTK-m framework, which has been developed to address the rapidly changing landscape of execution environments in HPC systems. The issue of portable performance across diverse architectures is of growing importance to simulation and experimental scientists across a wide set of disciplines. The growing compute and I/O imbalance in current and future HPC systems is causing a keen interest in scenarios where compute is moved to the data, as opposed to the traditional model where data are moved to the computational resources. Portability is extremely important in these use cases: a visualization code has to run with reasonable efficiency close to the data, regardless of the specific hardware that generated said data.

We have demonstrated the portable performance of our implementation on a set of typical workloads, across a representative set of vector fields, and on a diverse set of CPU and GPU hardware. We have shown that the behavior of our portable implementation agrees with previously published results on hand-coded GPU and CPU implementations. We have also compared the performance of our portable implementation to several hand-coded implementations on a variety of workloads and hardware configurations.

As stated previously, we have not performed any hand-tuning of the VTK-m backend to achieve portable performance. However, we believe that there are improvements that could be made to the VTK-m backends that would yield increased performance for particle tracing methods. We plan to explore and evaluate these aspects and balance them against the general performance of the VTK-m backend as a whole.

Finally, there are a large number of extensions to this work, including support for streamlines where particle trajectories are stored. Because of early termination of particles, it is unknown at runtime what memory resources are required for the particle trajectories. In the future, we plan to explore methods to efficiently support storing particle trajectories for streamlines. We are also planning on support for time-varying vector fields and the analysis techniques associated with these types of vector fields. Eventually, we would like to re-evaluate performance portability for both of these cases; due to the dynamic memory allocations (streamlines) and the increased read-bandwidth requirements (pathlines) we might find different effects. In that regard, we plan to study the impact of unified memory available on new GPUs for particle tracing in general, but also to handle the storing of particle trajectories for streamlines and pathlines.

In summary, we believe that the portable performance of our implementation makes it a fruitful platform for work in particle-advection based techniques.

CHAPTER IV

STEADY STATE PARTICLE ADVECTION ON GPUS AND CPUS

This chapter is based on a co-authored work which is in submission. I am the primary author for this chapter and I was the primary author for this chapter. In particular, I performed all coding and ran all experiments. Dave Pugmire and Hank Childs had input on experiments and paper direction, and Hank Childs provided editorial suggestions.

This study evaluates the benefit of using parallelism from GPUs or multi-core CPUs for particle advection workloads. We perform 1000+ experiments, involving four generations of Nvidia GPUs, four CPUs with varying numbers of cores, two particle advection algorithms, many different workloads (i.e., number of particles and number of steps), five different data sets, and, for GPU tests, performance with and without data transfer. The results inform whether or not a visualization developer should incorporate parallelism in their code, and what type (CPU or GPU).

4.1 Introduction

Particle advection, i.e., displacing a massless particle according to a vector field, is a foundational operation for flow visualization. This operation is carried out by calculating particle trajectories by a series of “advection steps.” Each advection step involves evaluating a vector field at one or more locations and then solving an ordinary differential equation. Some flow visualization techniques require many particles, many advection steps per particle, or both. As a result, particle advection-based flow visualization can be very computationally expensive, making interactivity difficult.

Parallel processing is a key approach reducing long execution times. That said, this very simple question — “how much speedup should I expect to get if I enhance my visualization software to use parallelism?” — has a surprisingly complex answer. The followup question “which type of parallelism should I use?: CPU or GPU" also is non-obvious. On the one hand, GPUs provide significant computational resources, making them potentially very useful for particle advection problems. That said, it is quite difficult to achieve peak performance on a GPU, especially for data intensive operations. Further, particle advection-based flow visualization includes diverse use cases which can lead to varying performance characteristics and varying speedups across GPU architectures. On the other hand, while multi-core CPUs often do not have the raw FLOPS of a GPU, they can compare favorably to a GPU either because of faster individual cores or because of direct access to data (i.e., no data transfers).

In response, we consider five related research questions on particle advection performance:

- **RQ1**: How much speedup will a GPU provide over a serial CPU implementation? How much does individual GPU architecture matter?
- **RQ2**: How much speedup will a multi-core CPU provide over a serial CPU implementation? How much does CPU concurrency matter?
- **RQ3**: When given the opportunity to use either a GPU or a multi-core CPU, which should a visualization developer choose?
- **RQ4**: What is the impact of the data set on particle advection performance?
- **RQ5**: What are the trends in performance by using newer hardware?

The main outcome of this study is informing visualization developers whether parallelism will speed up their particle advection workloads, and, if so, to what extent. That said, the space of possible experiments is quite large and, to maintain an achievable scope, we introduce two boundaries. For the first boundary, our study is targeted at visualization software running on desktop machines, and does not consider supercomputers. Desktop machines are more widely accessible, and they increasingly have general-purpose computing environments on their GPUs (CUDA, OpenCL, etc.) and also significant parallelism via CPU cores. Further, while we do not run distributed-memory experiments on supercomputers, our findings also have implications for this environment. For the second boundary, our study focuses exclusively on steady-state flow. Steady-state flow advection is a common use case, and particularly common for the workloads that consider many advection steps (i.e., the workloads that benefit most from parallel processing). That said, our findings again have implications beyond our scope, and we consider how our findings apply to the unsteady-state case in our conclusions.

4.2 Related Work

Shared memory parallelism refers to using the parallelism available on a single node where the data resides in the main memory and is shared by all the parallel elements. Traditionally shared-memory parallelism has been possible using multi-core CPUs, and more recently, GPUs. There has been a significant body of particle advection works that focuses on using shared memory parallelism to improve the performance of related flow visualization algorithms.

Some past works have investigated the use of shared-memory parallelism for particle advection in the context of distributed-memory parallelism, i.e., MPI-hybrid parallelism. Camp et al. proposed two different algorithms for particle advection on large data that used multi-core CPUs for shared-memory parallelization [32]. The multiple cores of the CPU were used to perform particle advection, I/O operations, and communication between nodes. Camp et al. [33] later extended their work to use GPUs and compared the performance of GPUs to multi-core CPUs. Their findings reveal that the CPUs performed better for certain workloads where there are fewer particles or where the duration of advection is long. However, the GPU was able to perform better for the other workloads. Childs et al. [39] compared various GPUs and CPUs to understand the relationship between the execution devices and the execution time. They made two key observations, 1) CPUs were better for medium to long duration of advection, and 2) for many cases with overall short execution times CPUs matched or outperformed the GPUs. Jiang et al. used the multiple threads of CPUs to perform I/O operation to hide the I/O latency and improve particle advection performance [**?**]. Hentschel et al. [68] studied the benefits of using SIMD extensions to achieve better performance for particle advection. They packed spatially close particles together to use SIMD extensions efficiently by improving the spatial locality of memory accesses. They reported a performance improvement of 5.6× over a baseline implementation.

GPUs have gained a lot of popularity as general-purpose computing devices over the last decade because of specialized tools like Nvidia's CUDA library [98]. And as of lately, libraries like Kokkos [45], RAJA [10], VTK-m [95] allow users to write C++ code that runs on GPUs without the user requiring much expertise, making it easy to write parallel applications. GPUs offer excellent parallelism, given that there is enough work that can be efficiently parallelized. Particle advection lends itself to parallelization easily as each particle can be advanced independently, making it embarrassingly parallel. That said, parallelism is limited to only the particles, since each advection step for a particle depends on the result of the previous step.

Past studies have used GPUs for particle advection to perform interactive flow visualization. Krüger et al. investigated using GPUs to produce particle visualizations for a million particles at once [78]. Their strategy was to exploit the GPU's ability to perform particle advection and produce visualizations without having to move data between the GPU and the host. The authors demonstrated the efficacy of their system for unsteady-state particle visualizations and also for 3D steady-state visualizations like streamlines and stream ribbons. Bürger et al. extended the system of Krüger et al. to produce unsteady flow visualizations[27]. Their approach was to stream data to the GPU while the GPU was busy performing particle advection, such that the next data slice would be available when needed. Bürger et al. later demonstrated their system could efficiently recognize flow features using some measure of importance such as Finite Time Lyapunov Exponent (FTLE), helicity, vorticity, etc. [26]. Bürger et al. then demonstrated their system could render streak surfaces [24] by adaptively refining/coarsening the surface at interactive rates with the GPUs.

Pugmire et al. [110] implemented a platform portable particle advection solution using VTK-m [95]. They evaluated their implementation on multi-core CPUs and GPUs. Their demonstrated implementation can perform well against platform optimized particle advection solutions, demonstrating good portability. Their work has also been of crucial importance for many studies that investigate the efficiency of large scale MPI-hybrid parallel particle advection [18, 16, 19, 17]. Our work is different than that of Pugmire et al. as we focus on answering our research questions, where their focus was on demonstrating portable performance.

There have been many studies about the performance of particle advection at large scale by optimizing certain aspects for distributed particle advection. Operating in a distrubuted setting often demands data to decomposed into small blocks which are distributed among processes. Since particle advection is highly data dependant domain decomposition to ensure load balanced computation is important. Efficient domain decomposition can also help in avoiding unnecessary I/O and communication. To that end many studied have proposed schemes for data decomposition for particle advection based algorithms [37, 102, 100, 135] or for efficient I/O performance [75]. Another important aspect of distributed particle advection is work distribution and scheduling that leads to efficient use of parallelization and underlying hardware. To that end studies proposed new parallelization strategies [108, 99, 31, 90, 97] and demonstrate ways to use the whole spectrum of execution devices available using co-processing [58].

4.3 Experimental Overview

This section describes the setup for our experiments, which varied over the following parameters:

- Data sets: 5 options
- Advection workloads
    - Number of seeds: 5 options
    - Duration: 3 options
    - Seeding volume: 3 options
    - Algorithms: 2 options
- Hardware usage

    - ∗ GPU transfer modes: 2 options
    - ∗ Devices: 4 CPUs and 4 GPUs

These options create for a potential 5400 experiments total — 5 data sets × 90 advection workloads × 12 hardware options (4 for the CPU and 8 for the GPU with the transfer modes). That said, for each research question, we considered only a subset of the experiments, tailored to answer the question. For example, **RQ1** considered only 180 experiments while **RQ5** considered 864.

All experiments were run using particle advection modules implemented in VTK-m [95]. VTK-m takes a portably performant approach, i.e., a single code implementation can run efficiently in serial, in parallel on a CPU, or in parallel on a GPU. This approach has been demonstrated to produce code that runs as efficiently as CPU-specific code or GPU-specific code, with findings specifically considering particle advection [110] and also a meta-study considering nine different visualization algorithms [94].

The remainder of this section describes the options for our experiments in more depth.

4.3.1 Data Sets.

Four of the data sets for our experiments consisted of a vector field on a 512^3 uniform grid:

- ***Astro*** comes from a GenASiS simulation code [46] of a magnetic field simulation surrounding a solar core collapse that results in a supernova.
- ***Fusion*** comes from the NIMROD simulation code [123] of a magnetic field in a fusion tokamak device used to model the behavior of burning plasma.

- ***Fishtank*** comes from the NEK5000 simulation code [50] of a fluid flow inside a chamber when water of different temperatures is injected through a small inlet.
- ***Noise*** is a reference data set provided with the VisIt project [41]. The data set begins as scattered data, consisting of one hundred points in a volume. VisIt then constructs a vector field on a rectilinear grid by smoothly interpolating between the scattered data values.

The fifth data set, ***Zero***, consists of a single hexahedron with all vectors having zero magnitude. This data set served as a reference for cache performance.

RQ4 considers the impact of data set. Its findings show that data set is important, but not as important as other factors (hardware, workload). As a result, to simplify the analysis for **RQ1**, **RQ2**, and **RQ3**, their experiments only utilize the ***Noise*** data set. ***Noise*** was chosen because it has the least variability in number of steps, i.e., particles hit zero-velocity spots or exit the volume less often, making for more consistent results.

4.3.2 Workloads. In the context of this study, we define a particle advection workload to have four factors:

- **Number of seeds**: the number of particles placed in the volume. For this study, we considered five amounts: 100, 1000, 10,000, 100,000, and 1,000,000.
- **Duration**: the number of advection steps performed for each particle. For this study, we considered three amounts: 100, 1000, and 10000.
- **Seeding volume**: the size of the sub-volume where seeds are placed. For this study, we considered three seeding volumes: Small (i.e., all seeds are placed in a small region near each other), Medium, and Large (i.e., seeds are placed

randomly throughout the entirety of the data set). This factor is considered since small seeding volumes can have better cache coherency (especially in combination with short durations).

- **Algorithm**: how particle trajectories will be used. For this study, we considered two algorithms: particle advection and streamlines. Particle advection refers to simply advecting the particles to find their final position, which is useful for FTLE and some other advanced analyses. The streamline algorithm stores the resulting position of each advection step. These two algorithms were chosen because they demonstrate significant differences in the strain they place on the memory system.

4.3.3 Hardware Usage.

GPU Transfer Mode: This factor considers the difference in performance when data needs to be transferred to/from the GPU ("with transfer"), as opposed to when data is already in the GPU's memory ("without transfer"). We ran experiments of both types in our study, in the following way:

- **With Transfer:** time to transfer the vector field data to the GPU, the time to perform the algorithm (streamlines or particle advection), and the time to transfer the data back. Note that the amount of data transferred back is different based on algorithm: proportional to the number of seeds for particle advection and proportion to the number of steps (seeds times duration) for streamlines. Note that the amount of data transferred back is variable for streamlines

- **Without Transfer:** time to carry out the algorithm (streamlines or particle advection). In this scenario, the vector field data is already on the GPU, and the results are not transferred off the GPU.

Devices: Our experiments were run on four different machines which provided access to four different CPUs and four generations of Nvidia GPUs. Table 12 describes these configurations for these Machines. Alaska, Voltar, and Saturn (CPUs and GPUs 1, 2, and 4 respectively) are hosted at the University of Oregon, and Summit (CPU and GPU 3) is hosted at the Oak Ridge National Laboratory.

Table 12. The list of CPUs an GPUs that were used for the experiments in the chapter.

CPUs	GPUs
CPU1: 2 x Intel Xeon E5-1650 w/ 12 cores, 3.8 GHz, and 32 GB memory.	***GPU1:*** Nvidia Tesla K40C w/ 12 GB memory and double precision performance of 1.68 TFLOPS.
CPU2: 2 x Intel Xeon 6226R w/ 32 cores, 3.9 GHz, and 256 GB memory.	***GPU2:*** Nvidia Tesla P100 w/ 16 GB memory and a double precision performance of 4.7 TFLOPS
CPU3: 2 x IBM Power9 w/ 32 cores, 3.8 GHz, and 512 GB memory.	***GPU3:*** Nvidia Tesla V100 w/ 16 GB memory and a double precision performance of 7 TFLOPS.
CPU4: 4 x Intel Xeon 8367HC w/ 104 cores, 4.2 GHz, and 376 GB memory.	***GPU4:*** Nvidia Tesla A100 w/ 80 GB memory and a double precision performance of 9.7 TFLOPS.

4.4 Results

This section is organized around our five research questions, with Section 4.4.1 addressing **RQ1**, Section 4.4.2 addressing **RQ2**, Section 4.4.3 addressing **RQ3**, Section 4.4.4 addressing **RQ4**, and Section 4.4.5 addressing **RQ5**.

Table 13. The 270 configurations used to explore **RQ1**. 240 of these configurations were on a GPU and 30 served as baseline experiments on a serial CPU. However, 24 GPU experiments were unable to finish as the required memory exceeded the device memory; these experiments all involved streamlines with many advection steps.

Parameter	Value	Total
Data Sets	Noise	1
Seed Volume	Large	1
Seeds	All	5
Duration	All	3
Algorithm	Particle Advection, Streamlines	2
Hardware	CPUs (Serial), GPUs (w/o Xfer, w/ Xfer)	9

4.4.1 RQ1: How Much Speedup Will a GPU Provide Over a Serial CPU Implementation?.

Table 13 shows the parameters for this phase's experiments and Figure 11 shows the results for these experiments. This plot shows a wide range of outcomes — for some experiments a GPU can be over 100X faster than a serial CPU while other experiments show a serial CPU to be over 50X faster than a GPU. Across all experiments, however, the average GPU speedup is 6.14X compared to a serial CPU, with the following breakdown into ranges:

GPU Speedup	<1X	1X-4X	4X-16X	16X-64X	>64X
% of tests	20%	18%	28%	20%	14%

The following subsections analyze these results with respect to the importance of some of our test factors: number of steps, GPU architecture, algorithm, and memory transfer mode.

4.4.1.1 Effect from Number of Advection Steps.

This section considers the effects from the number of advection steps, i.e., number of particles and duration.

Number of particles is a dominant factor in speedup. This is an expected finding — parallelization occurs over particles, and having ten thousand particles

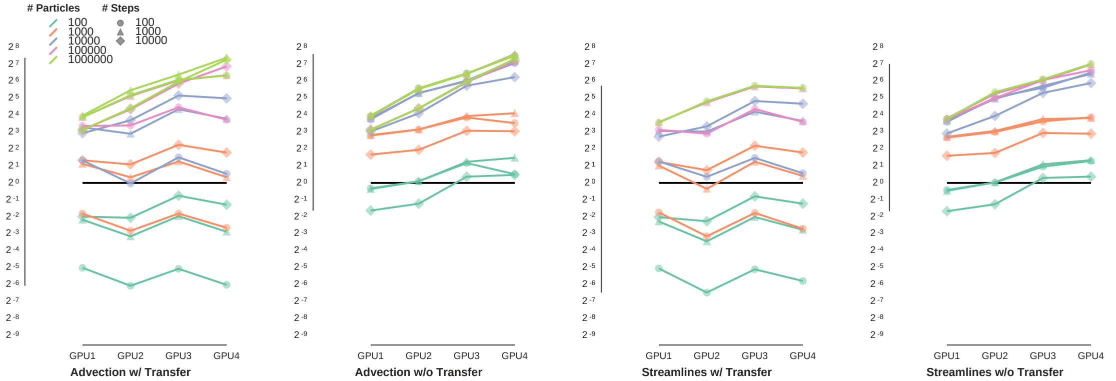


Figure 11. A chart showing GPU speedup compared to a serial CPU. For each of the four figures, the X-axis represents the GPU generation, older to newer from left to right. The colors represent the number of particles in the workload and the glyphs represent the duration of the workload.

or more allows all threads to be engaged. Focusing on the portion of Figure 11 devoted to advection without transfer, the speedups for ten thousand particles are nearly identical to those with one million particles. For the workloads with one million particles, the speedups are as low as 8X (on GPU1) and as high as 160X (on GPU4). Further, the workloads with ten thousand and one hundred thousand particles also show strong speedups. Looking at the other configurations in Figure 11, some ten thousand particle workloads are just as fast as million particle comparators. For others, the speedup is less, but still significant. For example, for “streamlines without transfer” on GPU4, the speedup with ten thousand particles is about 70X, while for one million particles it is over 120X. Across all configurations, the workloads with one hundred particles fare much worse, with speedups topping off at 2X, and many actually running slower on the GPUs. The workloads with one thousand particles perform better, with some seeing speedups of 8X, although some of these workloads are still slower on GPUs compared to a serial CPU.

Duration affects speedup less. For the workloads with ten thousand particles or more, the expected speedup does not change much as duration varies. For workloads with fewer particles, however, duration is a more significant factor. For example, for the "advection with transfer" case with one hundred particles, durations of 100 steps are 30X faster on a CPU while durations of 10000 steps are merely 4X faster on the CPU. In all, the effect of duration is only significant for workloads where GPUs provide little-to-no value.

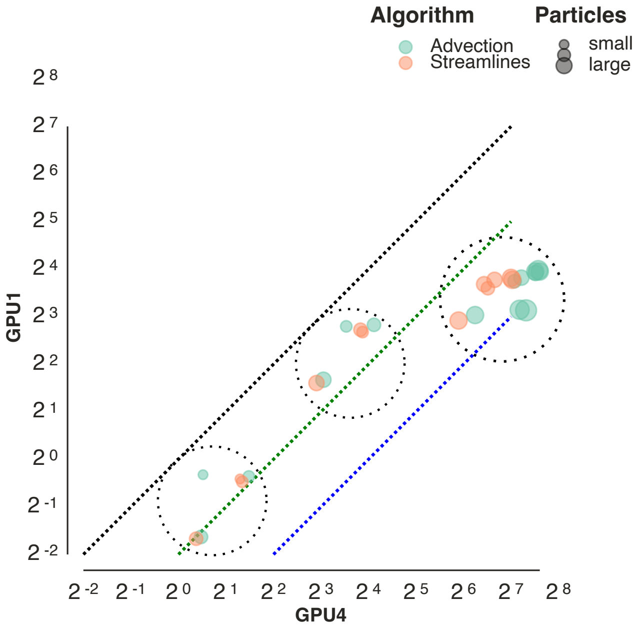


Figure 12. Scatter plot of speedups achieved for GPU4 (Ampere) versus GPU1 (Kepler) for workloads with no data transfer. If a given workload had a 30X speedup on GPU4 and a 6X speedup on GPU1, then that workload would be plotted at (30, 6) in this figure. The dotted lines show relationships between GPU1 and GPU4: the dotted black line shows where performance between the two GPUs is equal, the dotted green line shows where GPU4 is 4× faster than GPU1, and the dotted blue line shows where GPU4 is 16× faster than GPU1. Finally, the three dotted circles indicate three clusters of similarly performing experiments.

4.4.1.2 ***Effect of GPU Architecture.*** Figure 11 shows the expected result that newer GPUs are able to offer better performance. For the workloads with the most advection work, each newer generation of GPU provided an improvement in performance, especially in cases where memory transfers were not considered. The newest GPU (GPU4) provided a maximum speedup of 160X in case of particle advection and 130X in case of streamlines. The oldest GPU (GPU1) provided a maximum speedup of 16X in all cases.

Figure 12 plots the speedups for these extreme GPUs in our study: Ampere (GPU4) versus Kepler (GPU1). This plot shows three distinct clusters:

1. The first cluster contains workloads where neither GPU1 nor GPU4 offered any improvements over serial CPU. These workloads generally have a small amount of work to do. That said, GPU4 still performs 4× better compared to GPU1 for these workloads.

2. The second cluster contains workloads where both GPU1 and GPU4 offer significant speedups over serial CPU. Once again, GPU4 is only $2 - 4\times$ faster than GPU1.

3. The third cluster contains workloads with a bigger disparity between GPU4 and GPU1 (much larger than 4X): ~128X speedups for GPU4 versus only ~8X for GPU1. This is where the majority of our workloads fall, and thus reflects the most common outcome within our corpus of tests. The best speedups were achieved by the workloads that only advected particles and did not generate streamlines, which leads into the next section on algorithm effects.

These clusters are revisited in Section 4.4.4 which looks at hardware trends.

4.4.1.3 Effect of Algorithm. In the context of our performance study, there are two main effects due to algorithm. First, the streamline algorithm stores each particle position (12 bytes of position data for every advection step), which can stress the memory system. Second, the output of the streamline algorithm is much larger than particle advection, and so the configurations where data is transferred back to the CPU can potentially face bottlenecks. Figure 11 illustrates the impact of each effect. First, the second and fourth sub-figures of Figure 11 show the experiment results without transfer, i.e., it shows the differences solely due to storing more particle positions. The average speedup for streamlines (fourth sub-figure) is 10.45X, while the average speedup for advection (second sub-figure) is 12.28X, i.e., streamlines' extra memory stressors cause a 17% slowdown. (Note that some streamline experiments could not complete due to exceeding memory, and the corresponding advection experiments were removed for this analysis.) Next, the the first and third sub-figures of Figure 11 inform the effects of transfer. For the experiments involving transfer, the average speedup for streamlines (third sub-figure) is 1.92X, while the average speedup for advection (first sub-figure) is 2.23X. The gap between the two algorithms has narrowed from 17% to 16%, i.e., the fixed cost of transferring data set causes them both equal slowdown and the extra memory stressors for streamlines becomes slightly less pronounced. Finally, the averages presented in this analysis are geometric means, which help with interpreting behaviors that range between large speedups and slowdowns. Repeating the analysis with arithmetic means gives 13.25X, 37.78X, 9.88X and 27.32X for the four sub-figures. While these numbers are skewed higher by the experiments with many advection steps, the same trends hold: without transfer has a 38% slowdown for streamline memory stressors, and adding transfer times narrows to 34%.

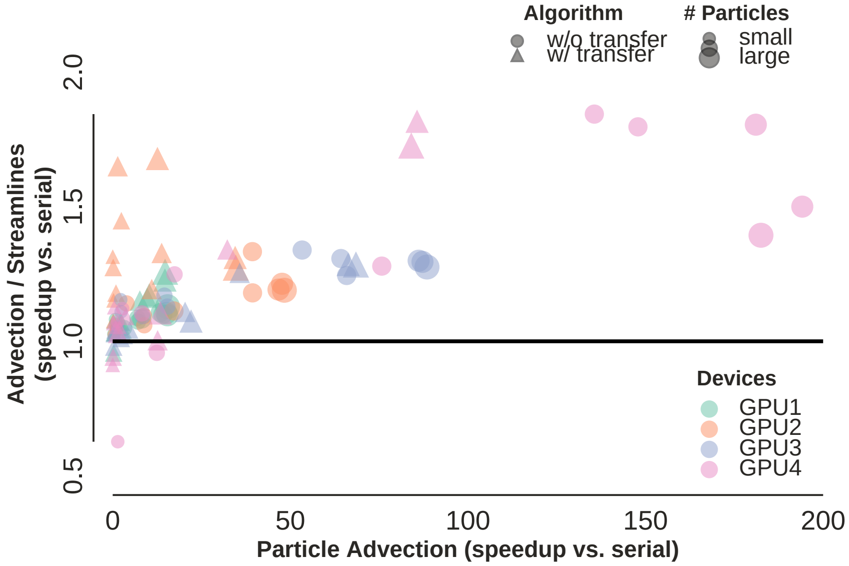


Figure 13. A scatter plot showing the differences between the particle advection and streamline algorithms. The X-axis represents the speedup achieved by a certain workload for the particle advection workload, and the Y-axis represents how much faster the particle advection algorithm executed compared to the streamline algorithm. A glyph at (X, Y) indicates the particle advection algorithm running on a GPU achieved a speedup of X times, and that its speedup was Y times better than the streamline algorithm with the similar workload. The glyph color indicates the GPU device type, the glyph shape indicates whether or not data transfer was involved, and the glyph size indicates the number of particles (which informs how many GPU cores could be engaged).

Finally, Figure 13 shows a scatter plot considering speedups for the two algorithms on GPUs compared to a serial CPU. This figure has several findings. First, streamlines consistently achieve speedups within a factor of two of advection. Second, the ratio between streamline speedup and advection speedup appears to get larger as the overall speedup improves. In other words, in the cases where the speedup is great (i.e., many advection steps), streamlines fall off the pace somewhat, due to stresses on the memory system. Third, the GPUs perform differently. In particular, GPU2 has worse streamline performance than the other hardware architectures.

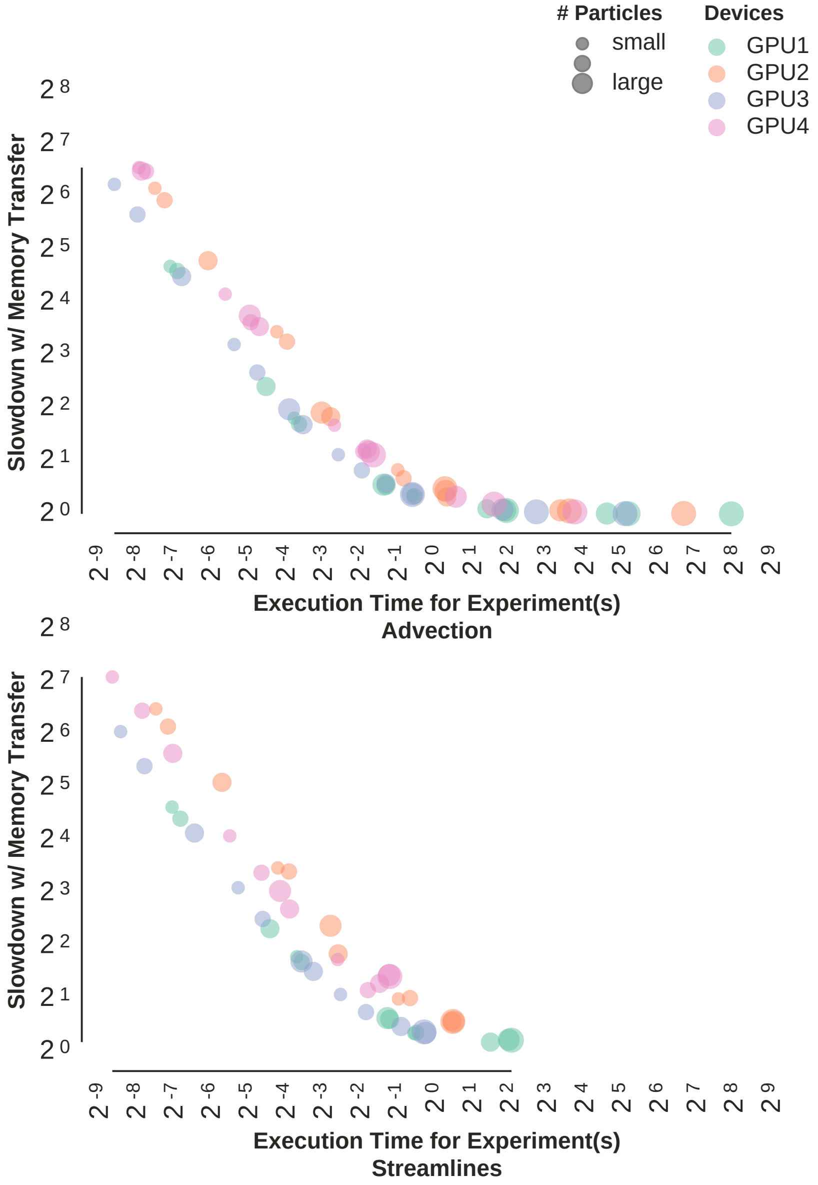


Figure 14. Plotting the slowdown for using memory transfer as a function of execution time. The figure is split into two, with advection experiments in the top figure and streamline experiments in the bottom figure. A glyph at (X, Y) means that a given workload took X seconds to execute without transfer and that workload was Y times slower when running with transfer. There are no glyphs for streamlines with execution times greater than four seconds, since those experiments consistently exceeded GPU memory.

4.4.1.4 Effect of GPU Transfer Mode. Figure 14 shows the effect of GPU transfer mode, i.e., if the data starts on the CPU, then how much effect is there to transfer the data to the GPU and back? In the “with transfer” experiments, the vector field was always transferred to the GPU, as were the starting seed positions. The data retrieved differed based on algorithm: either every position of every step (streamlines) or just final particle position (advection). This overhead was significant, as no experiment that involved transfers went faster than 0.18s. As a result, the overhead dominates the left portion of both figures — when the runtime without transfer is fast, then the slowdown is proportional to the data transfer time. For example, for streamlines on GPU4 with 10000 particles going 100 steps, the time without transfer is 0.007s and with transfer is 0.372s for a transfer slowdown of 50X. Both plots show an inflection point around execution times of 0.5s, when the data transfer overheads are more amortized. That said, relatively few streamline experiments are able to benefit from this amortization, since the streamline experiments that ran large numbers of advection steps to exceed 0.5s often ran out of memory. In other words, streamlines with memory transfer was almost always a poor idea in our set of experiments — with little work, the transfers dominated while with significant work, the experiment could not complete. Advection, on the other hand, showed significant benefit for the highest workloads.

Summarizing, the main findings from this analysis are: (1) small workloads perform poorly due to data transfer overhead, (2) few workloads perform well with the streamline algorithm due to data transfer overhead, and (3) large workloads can perform well with the advection algorithm.

4.4.1.5 Synthesis. Table 14 synthesizes the findings from the previous section. Each of the four factors (# of advection steps, architecture, transfer mode, algorithm) significantly affects performance:

Table 14. The speedups over serial execution achieved by the two algorithms for the different workloads while using GPU1 and GPU4 (denoted G1 and G4 in the table) with both transfer modes. Each value represents the average of all workloads that used the specified number of steps. Note the 10^9 workload consists of 1M particles for 1K steps and 100K particles for 10K steps, while the 10^{10} workload consists of only 1M particles with 10K steps. The shorter duration (1K) workload ran faster, resulting some apparent slowdowns between the 10^9 and 10^{10} cases. In actuality, the 10K experiments did increase speedup when going from 100K particles to 1M particles.

# of Steps	Advection				Streamlines			
	w/ X		w/o X		w/ X		w/o X	
	G1	G4	G1	G4	G1	G4	G1	G4
$< 10^5$	< 1	< 1	< 1	< 1	< 1	< 1	< 1	< 1
10^5	< 1	< 1	2	5	< 1	< 1	2	5
10^6	< 1	< 1	3	14	< 1	< 1	2	10
10^7	6	8	8	60	5	7	8	39
10^8	11	61	12	138	9	38	11	100
10^9	11	139	11	166	X	X	X	X
10^{10}	8	155	8	160	X	X	X	X

- GPUs were not useful for many workloads with small numbers of steps, although the threshold for when they became useful varied based on GPU transfer mode.
- For workloads doing the particle advection algorithm and many advection steps, the GPU transfer mode becomes less relevant.
- The streamline algorithm is a little slower than the particle advection algorithm in all cases, but the magnitude of effect is not as big as the other factors. That said, the streamline algorithm is not viable for very large numbers of steps.
- The improvements in hardware architecture (GPU1 to GPU4) make an impact (4X-20X) in almost all cases where a GPU can outperform a serial CPU. The most notable cases where the change in hardware architecture does

not make an impact is with the 10^7 workloads with data transfer. In these cases, the data transfer time is large enough that the increased computational power does not significantly affect speedup. Larger workloads do benefit from the increased computational power from GPU4, while smaller workloads are affected by transfer times to the point that a serial CPU is preferable.

Table 15. The 150 configurations used to explore **RQ2**. 120 of these configurations were on multi-core CPUs and 30 served as baseline experiments on a serial CPU. The largest serial CPU streamline experiment failed so we have excluded the 4 respective multi-core counterparts, leading to a total of 116 multi-core experiments.

Parameter	Value	Total
Data Sets	Noise	1
Seed Volume	Large	1
Seeds	All	5
Duration	All	3
Algorithm	Particle Advection, Streamlines	2
Hardware	CPUs (Serial, Multi)	5

4.4.2 RQ2: How Much Speedup Will a Multi-core CPU Provide Over a Serial CPU Implementation?.

Table 15 shows the parameters for the experiments we conducted for this phase and Figure 15 plots the speedup for multi-core CPUs using both the advection and streamlines algorithms, and shows the expected result that newer CPUs with more cores perform better than older CPUs with fewer cores. Compared to GPUs experiments, this plot shows relatively narrower range of outcomes — multi-core CPUs perform better than serial in all but two cases. These outcomes spanned a spectrum from 6X to 64X. Across all experiments, the average CPU speedup is 13.91X compared to a serial CPU, with a breakdown into ranges as follows:

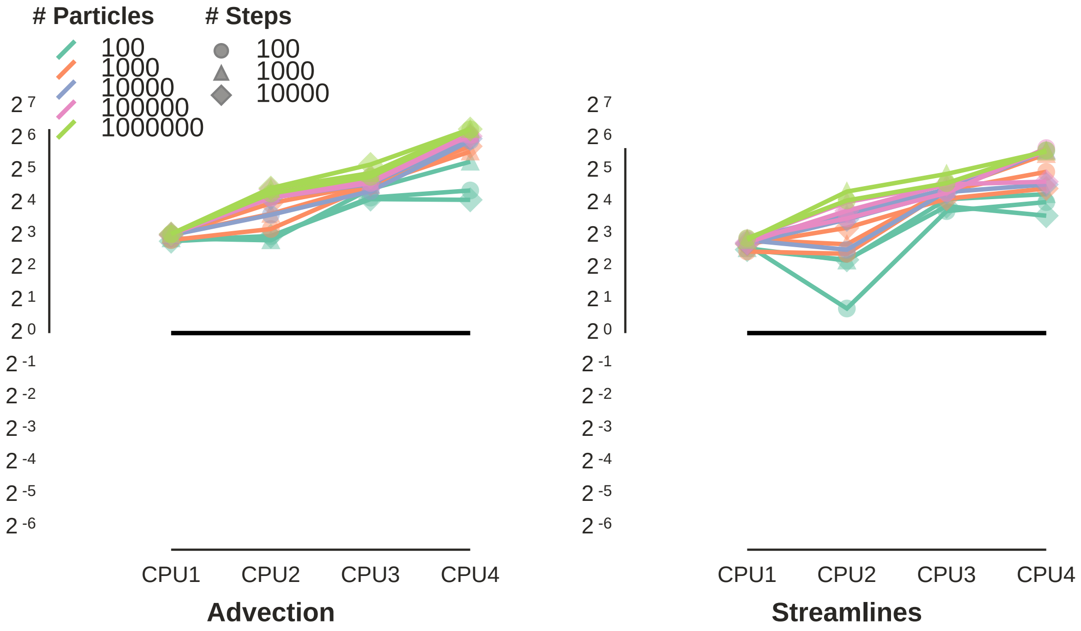


Figure 15. A chart showing CPU speedup compared to a serial CPU. For both subfigures, the X-axis represents the CPU generation, older to newer from left to right. The colors represent the number of particles in the workload and the glyphs represent the duration of the workload.

CPU Speedup	<1X	1X-4X	4X-16X	16X-64X	>64X
% of tests	0%	1%	41%	50%	8%

In all cases, the multi-core CPUs fall short of their maximum possible speedup, e.g., CPU4 (104-core Xeon) achieves ~70X speedups instead of 104X speedups. That said, this level of speedup is consistent with previous multi-core scaling studies.

Focusing on the results from the advection algorithm, all CPUs demonstrated the same behavior. CPU1 is able to consistently provide similar speedup of 8X for all workloads, while the other CPUs demonstrate a spread in terms of their speedups based on the number of particles in the experiments. However, the spread between the best and worse speedup using any CPU is smaller that the corresponding GPU experiments; the highest spread for CPUs is for CPU4 with a spread of 4.5X between the best and the worse experiment, the spread for the corresponding GPU, GPU4, is 140X.

In terms of the parallel performance efficiencies, CPU2 and CPU3 (> 75%) performed better than CPU4 and CPU1 (66%). For all CPU2 and CPU3 experiments, speedup increased for both, i.e., speedup increased when increasing the number of particles or increasing their duration. While CPU1 is able to offer its best performance even for the workloads of lower magnitude due to minimal overhead of using threads, the limited number of threads and slower memory prevents it from performing better for workloads with a greater magnitude. On the other hand, CPU4, which offers a lot of parallelism, suffers from an initialization cost for smaller workloads (thread launch) and poor memory accesses (across NUMA regions) for the larger ones.

The streamline results from Figure 15 shows the effects of memory accesses from storing each advection step. The 32-core CPU2 and CPU3 are both slowed by ~2X. The 104-core CPU4 is affected more dramatically, as performance starts dropping as the number of advection steps increases, sometimes falling below the 32 core CPUs. In all, memory effects clearly denigrate the streamline algorithm's performance. Finally, Figure 16 shows a comparison between the 12-core CPU1 and the 104-core CPU4. This figure emphasizes that the behavior is varying across these architectures. It shows that the CPU1 has very consistent performance, while CPU4 spans the spectrum from doing ~12X faster to performing ~70X faster. CPU4 is able to offer better scalability as the workload increases as each core gets substantial work relative to overhead of assigning work. Eventually, for streamlines,

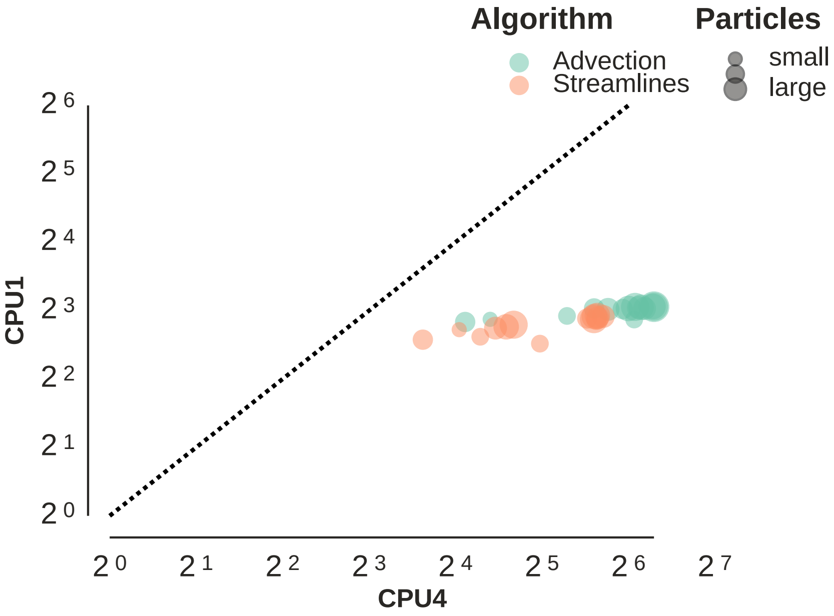


Figure 16. This scatter plot compares the performance of a 104-core Xeon with a 12-core Xeon. If a workload was 64X faster than a reference serial implementation on the 104-core Xeon and 6X faster on the 12-core Xeon, then a glyph would be placed at (64, 6).

the performance is capped at at 40X, and for particle advection, the speedups can exceed 64X.

4.4.2.1 Takeaways For CPUs. Table 16 presents the speedups for CPUs while considering all the factors that impact the performance discussed in this section. Since using multi-core CPUs is always helpful, this table acts a lookup for users to determine the extent of speedups they can achieve using multi-core CPUs. For both algorithms, multi-core CPUs are able to achieve the best performance once the workload has a total number steps $\geq 10^7$. CPUs with a lower number of total cores are able to achieve their best performance even for workloads of lower magnitudes, however, newer CPUs with a lot many cores show a steady increase in speedups with increases in the workloads.

Table 16. The speedups over serial execution achieved by the two algorithms for the different workloads while using CPU1 and CPU4. This table can be used a a lookup to estimate the speedup that can be expected for executing a certain workload.

Workload	Arch.	10^4	10^5	10^6	10^7	10^8	10^9	10^{10}
Advection	C1	7	7	7	8	8	8	8
	C4	21	51	37	64	70	72	78
Streamline	C1	6	5	6	7	7	7	X
	C4	16	24	30	38	38	35	X

Table 17. The 360 configurations used to explore **RQ3**. 120 experiments were run on multi-core CPUs, 120 experiments were run on GPUs and did not involve data transfers, and the final 120 experiments were run on GPUs and involved data transfers.

Parameter	Value	Total
Data Sets	Noise	1
Seed Volume	Large	1
Seeds	All	5
Duration	All	3
Algorithm	Particle Advection, Streamlines	2
Hardware	CPUs (Multi), GPUs (w/o Xfer, w/ Xfer)	12

4.4.3 RQ3: How Does GPU Performance Compare to Multi-Core CPU Performance?.

Table 17 shows the parameters for the experiments we conducted for this phase and Figure 17 compares multi-core CPU performance with GPU performance for each of our four machines. Of note, the CPU power and GPU power appears to be somewhat balanced across the machines, with the 12-core Xeon paired with the Kepler GPU (CPU1 and GPU1), the 32-core Xeon paired with the Pascal GPU (CPU2 and GPU2), the 32-core Power9 paired with the Volta GPU (CPU3 and GPU3), and the 104-core Xeon paired with the Ampere GPU (CPU4 and GPU4). In terms of results, an overwhelming trend is the imbalance in the range of outcomes — some multi-core CPU experiments are

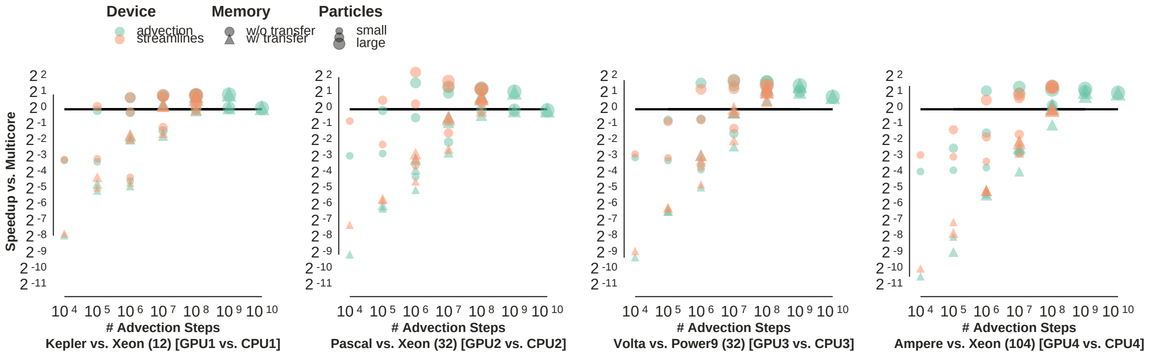


Figure 17. Comparison of GPU speedups for particle advection over their multi-core CPU counterparts with log scale in the Y-axis. This enables the identification of cases where the GPUs perform worse or better than a multi-core CPU. The horizontal line at $Y = 1$ is where the GPU performance equals that of a multi-core CPU. All data points below the line indicate slower GPU performance for an experiment.

as much as 1000X faster than their GPU counterparts, but GPU experiments are never more than 4X faster than their multi-core CPU counterparts.

For experiments without data transfer, the key factor in whether the GPU or CPU will be faster is the number of particles advected. For the most part, if 10,000 or more particles are advected, then the GPU is faster (since all GPU cores can be engaged), and if 1,000 or fewer particles are advected, then the CPU is faster (since the GPU cores cannot all be engaged). Experiments involving data transfer and streamlines are almost always faster on the CPU. The only exceptions involve cases with 100M advection steps, and even then speedups were only modest ($< 2\times$). One reason is that GPUs can only outperform multi-core CPUs when there is significant work, but, for the streamline algorithm, the amount of work needed to offset transfer costs exceeds GPU memory.

Finally, Figure 18 shows results when comparing the extreme of each architecture: best CPU (CPU4) vs worst GPU (GPU1), best CPU (CPU4) vs best GPU (GPU4), worst CPU (CPU1) vs worst GPU (GPU1), and worst CPU (CPU1)

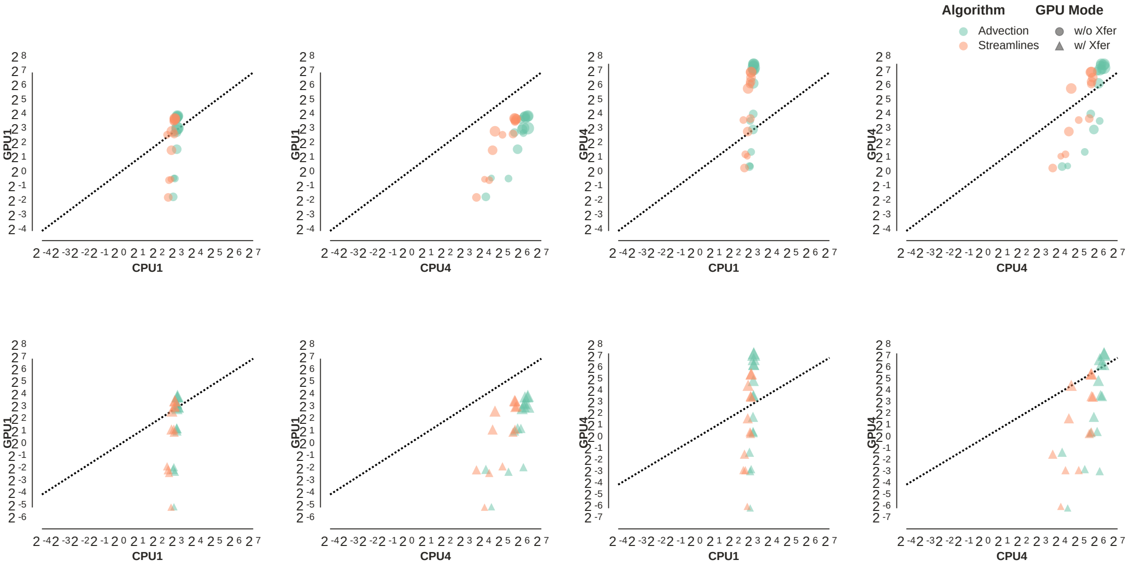


Figure 18. A small multiples chart comparing speedups for GPUs and multi-core CPUs compared to a serial CPU. Each figure contains a scatter plot with CPU results along the X-axis and GPU results along the Y-axis. Each point in the scatter plot corresponds to a workload; when comparing to serial CPU times, if a workload was 30X faster on a multi-core CPU and 15X faster on a GPU, then a glyph would be placed at (30, 15). The overall layout of the small multiples chart is 4x2. The two rows correspond to data transfer, with top row plotting without data transfer and the bottom row plotting with data transfer. The four columns go through the combinations of best and worst CPU and GPU. The worst GPU in our study, GPU1 (Kepler), is in the left two columns, while the best GPU in our study, GPU4 (Ampere), is in the right two columns. The worst CPU in our study, CPU1 (Xeon 12 cores), is in the first and third columns, while the best CPU in our study, CPU4 (Xeon 104 cores), is in the second and fourth columns.

vs best GPU (GPU4). CPU4 beat the worst GPU1, and often by significant amounts, in all but a few configurations. CPU4 was also able to beat the best GPU4 in most configurations. When considering experiments with data transfer, the CPU4 is able to either beat or keep up with GPU4 in almost all cases, with the only exception being those with large workloads. The worst CPU (CPU1) beat the worst GPU (GPU1) only for smaller workloads. When comparing the worst CPU

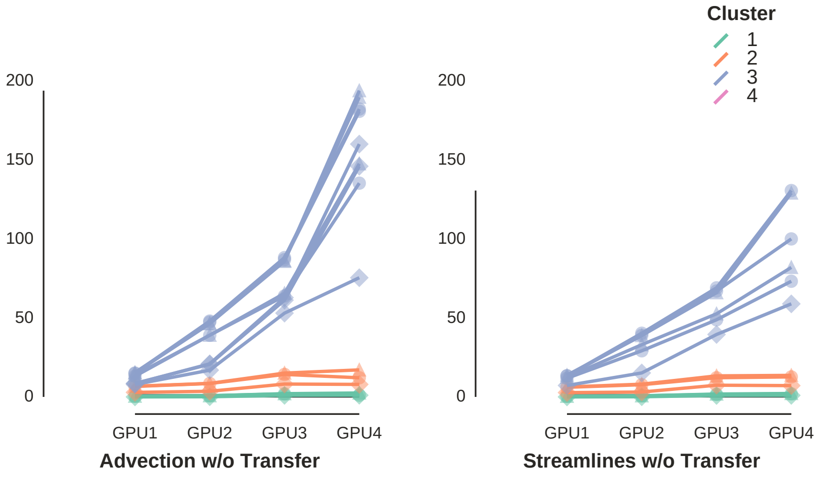
(CPU1) and the best GPU (GPU4), GPU4 was substantially faster for almost all of the cases losing only in the cases that involve small workloads.

Table 18. The ratio of the GPU and CPU speedups achieved by the two algorithms for the different workloads while using GPU1 with CPU1 and GPU4 with CPU4 with both the memory modes. The entries are represented as (GPU1/CPU1) / (GPU4/CPU4) in the table. This table can be used a decision lookup of whether to use the GPU or the CPU for executing a certain workload.

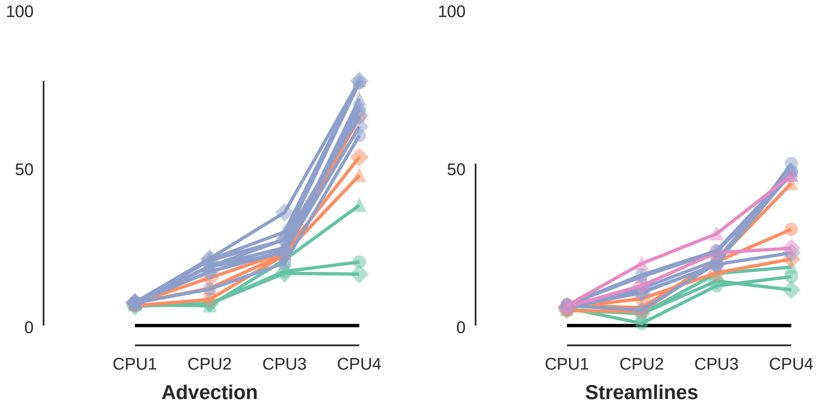
Workload	Advection		Streamlines	
	w/ X	w/o X	w/ X	w/o X
$< 10^7$	< 1	< 1	< 1	< 1
10^7	$< 1/1.0$	< 1	$< 1/1.1$	$< 1/1.02$
10^8	1.4/1.5	$< 1/1.9$	1.3/1.5	$< 1/2.58$
10^9	1.4/1.4	1.9/2.2	X	X
10^{10}	1.0/1.0	1.9/2.0	X	X

4.4.3.1 Takeaways for GPUs and CPUs. Table 18 presents the factor of speedups (over serial) achieved by GPUs over their multi-core CPU counterparts. A factor greater than 1 means that the GPU performed as many times better than their comparator CPU. The table also acts as an easy lookup for users to determine when GPUs will be better than CPUs for a particular workload. In general, for workloads below the total number of steps being less than 10^8, GPUs should not be used as they perform worse than the comparator CPUs. The range where GPUs are helpful for generating streamlines is very narrow, as they need significant amount of work to offset the costs of memory operations but are not able to support workloads which have memory requirements higher than the available GPU memory.

4.4.4 RQ4: What Are the Trends in Performance by Using Newer Hardware?. For GPUs, the three clusters identified in Section 4.4.1.2 inform hardware trends both for the last decade of GPUs and looking forward. Some workloads have so little work that GPUs are not very useful. The



(a) GPU trends



(b) CPU trends

Figure 19. Figure plotting the speedups achieved by a workload by the different generation of GPUs and CPUs for the two algorithms. (a) shows GPU scaling similar to Figure 11, and (b) shows CPU scaling data similar to Figure 15. Both of these figures plot the Y-axis without the log scale unlike the reference figures. The plots are colored by the cluster they are categorized in in Figure 12. Cluster 1 represents the poorest performing cluster and Cluster 3 represents the best performing cluster. Cluster 4 represents the experiments that finished on the CPUs but not on the GPUs due to limited memory.

improvements from GPU1 to GPU4 cause these workloads to cross the threshold of outperforming a serial CPU when there is no data transfer involved. It would be surprising if future GPUs become so powerful that they become relevant for these small workloads. Other workloads have modest speedups compared to a serial CPU. GPU1 had 3X to 8X speedups on these workloads and GPU4 improved these to 8X to 20X. That said, given the performance of multi-core CPUs, GPUs are unlikely to be the best choice for these workloads – if a visualization programmer is choosing between implementing a multi-core CPU code base and a GPU code base to address these workloads, they should likely choose a multi-core CPU approach. The final cluster of workloads is the one where recent improvements in GPUs have made the most difference, and future GPUs are likely to improve further still. GPU1 had speedups of 8X to 16X, but GPU4 improved these to 64X to 128X. For the most part, these are the workloads where GPUs outperformed the multi-core CPU. Further, given hardware trends (more cores, better memory infrastructure), future GPUs are likely to improve these speedups further.

Finally, Figure 19 presents the scaling behaviors of all the GPUs and CPUs considered in our study. It combines the clusters identified in Figure 12 and uses it to enumerate the experiments in Figures 11 and 15. This figure reveals insights into advances in GPUs and CPUs and informs their performance based on two factors, the workload and the algorithm. Focusing on workload, For the largest ones (identified by Cluster 3), newer GPUs and CPUs are able to offer increasing speedups. This is due to the fact that newer CPUs and GPUs have more numerous cores and each core is able to perform much better than the previous generations. These additional cores help workloads that are able to saturate all the cores of the GPUs and the CPUs. For the smaller workloads (identified by Clusters 1 and

2), however, the newer generations of GPUs did not offer significant performance improvements over the previous generations. Newer CPUs are able to offer modest speedups for the workloads in Clusters 1 and 2. Focusing on algorithm, newer GPUs demonstrate similar same behaviors of scaling for both the particle advection and streamlines, i.e., for both the algorithms workloads in Cluster 3 are able to scale very well and workloads in Clusters 1 and 2 are not. This trend is not consistent for CPUs. Newer generations of CPUs are able to behave similarly for particle advection, but demonstrate much smaller gains for streamlines, many times even performing worse than previous generations. However, there's still value in using CPUs for streamlines as very large workloads (in our case, total steps $\geq 10^8$) GPUs cannot be used because of very high memory requirements, while CPUs can still offer decent speedups. These very high workload cases are identified by Cluster 4. Hence, the findings can be summarized as:

- Newer generations of GPUs are able to scale much better for both particle advection and streamlines only for workloads that are able to use all the cores of the GPU.
- Newer generations of CPUs are much better at scaling or particle advection, but not for streamlines. However, they can still support very large workloads.

4.4.5 RQ5: How Does Data Set Impact Performance? .

This section considers the effect from data set on performance on GPUs and multi-core CPUs. Table 19 shows the parameters for the experiments we conducted for this phase. Data set can affect performance in two fundamental ways: caching and divergence. With respect to caching, some data sets may attract particles to key regions, potentially increasing cache performance, while other data sets

Table 19. The configuration for experiments that were executed to answer **RQ5**. Based on the parameters used, a total of 936 experiments were considered to answer this question of which 72 were run with Zero data set and served as a baseline ("Zero" data set used only one seeding volume).

Parameter	Value	Total
Data Sets	All	5
Seed Volume	All	3
Seeds	{100, 10000, 1000000}	3
Duration	All	3
Algorithm	Particle Advection	1
Hardware	CPUs (Multi), GPUs (w/o Xfer)	8

may move particles throughout the region uniformly, potentially decreasing cache performance. With respect to divergence, the fundamental issue is early termination, i.e., if a particle is supposed to advect for a duration of N steps, but it stops after K steps, where $K < N$. This early termination can happen because a particle entered a zero-velocity region (making further steps unnecessary) or because it exited the data set's spatial domain (making further steps impossible). In the context of a parallel architecture, the effect of early termination is that some threads will be asked to do asymmetric work — threads assigned particles that terminate early will perform fewer advection steps — which can create performance degradation due to divergence.

Measuring the effect of data set on performance is non-trivial. The execution time for a given workload on a given architecture reflects a combination of factors: clock speed, number of steps taken, and hardware efficiency (i.e., caching and divergence). Since our interest is on the hardware efficiency changes due to data set, we normalize our analysis with respect to clock speed and number of steps taken. We did this normalization by using a reference data set, which refer to as "Zero." This data set consists of a single cell with velocity value $(0, 0, 0)$ everywhere in the cell. For a workload W, a hardware architecture H, and a data set D, our analysis used the following terms:

- $N_{W,H,D}$: the number of advection steps performed for workload W, hardware architecture H, and data set D.
- $T_{W,H,D}$: the execution time for workload W, hardware architecture H, and data set D.
- $A_{W,H,D}$: the average time per step for workload W, hardware architecture H, and data set D. This is calculated as $A_{W,H,D} = \frac{T_{W,H,D}}{N_{W,H,D}}$.
- $Norm_{W,H,D}$: the normalized time per advection step for workload W, hardware architecture H, and data set D. This is calculated relative to the Zero data set as $Norm_{W,H,D} = \frac{A_{W,H,D}}{A_{W,H,Zero}}$.
- $AggrNorm_{H,D}$: the aggregated time per advection step for hardware architecture H and data set D over all six workloads. This aggregation is performed with a geometric mean, i.e., $(\prod_{w=1}^{w=6} Norm_{w,H,D})^{\frac{1}{6}}$.

To understand the meaning of these terms, consider an example. If Workload W4, hardware architecture GPU2, and data set Astro, have $Norm_{W4,GPU2,Astro} = 1.2$, then the average step was 20% slower than for the Zero data set. Further, if $AggrNorm_{GPU2,Astro} = 1.4$, then then the average step was 40% slower than for the Zero data set over all workloads. The remainder of this section focuses on $AggrNorm$ values. That said, the individual $Norm$ values for each comparison (over the four data sets, eight hardware architectures, and six workloads) can be found in the supplemental material,

Table 20 contains the $AggrNorm$ values for each combination of hardware architecture and data set. There are two primary findings from this table: effects from hardware architecture and effects from data set. With respect to hardware

Table 20. $AggrNorm$ values for all combinations of hardware architectures and data sets. Each entry cell shows the aggregate slowdown (over six advection workloads) for a given data set compared to the $Zero$ data set on a given architecture. For example, the number ***1.08*** in the top left of the table means that the Fusion data set took an average of 8% longer than the Zero data set on Xeon 8 architectures.

	Fusion	Astro	Fishtank	Noise
CPU1	1.08	1.02	1.35	0.98
CPU2	1.35	1.13	1.85	1.13
CPU3	1.23	1.11	1.76	1.05
CPU4	1.12	0.99	1.61	0.93
GPU1	2.77	2.49	7.04	1.97
GPU2	3.13	2.73	8.15	1.97
GPU3	4.99	4.17	9.55	3.90
GPU4	3.73	3.03	9.58	2.35

architecture, the impact for CPUs is significantly less than that for GPUs. The largest $AggrNorm$ value for CPUs is 1.85, while the largest value for GPUs is 9.58. This means that our experiments showed at worst an 85% slowdown on CPU architectures, compared to a 858% slowdown for GPUs. Further, GPUs were almost always twice as slow compared to the Zero data set, while CPUs were able to run nearly as quickly in some cases. In short, caching and divergence affected the CPUs much less than the GPUs. While this is an expected outcome, the magnitude of the effect (1.85 vs 9.58) was surprising. Another finding for hardware was that the $AggrNorm$ values climbed on each subsequent generation of GPU, from an average of 3.12 on GPU1 to an average of 3.99 on GPU4. So while later GPUs offer higher performance, degradations due to data set effects become more prominent. With respect to data set, Table 20 informs the extent of slowdown due to data set. The Noise data set (which has few zero-velocity spots and does not regularly push particles outside its spatial boundary) is able to perform similarly to the single-cell Zero data set on CPUs, and performs better than the other data sets on

GPUs. The Fishtank data set is the clear worst performer, with GPU performance being approximately three times slower than the other data sets. Once again, while this type of effect is to be expected, we found the magnitude of this effect to be surprising. That said, more analysis is needed to figure out the cause of the slowdown: divergence, caching, or both.

Figure 20 and Table 21 isolate the effects from divergence. As stated earlier, divergence occurs because of early termination. Our solution is to modify our algorithm to not terminate these particles — if a particle hits a zero-velocity region, then the algorithm continues performance advection steps (and remaining in the same position) and if a particle exits the spatial domain, then we have it advect backwards in time (i.e., retrace the path from where it came). For ease of reference, we refer to these additional steps as "fake steps."

Table 21. Table demonstrating the relative slowdown in terms of time per step for different architectures when particles are not terminated at all. For example, for Xeon 8 CPU and Fusion data set the number ***1.06*** represents that the experiment is 1.06x slower than the ideal case.

	Fusion	Astro	Fishtank	Noise
CPU1	1.06	1.02	1.22	0.99
CPU2	1.14	1.05	1.42	1.11
CPU3	1.11	1.07	1.28	1.02
CPU4	1.02	0.97	1.26	0.94
GPU1	1.84	1.62	1.75	1.61
GPU2	1.54	1.46	1.78	1.48
GPU3	3.42	3.02	3.70	3.06
GPU4	2.28	2.07	2.62	1.90

Figure 20 plots the proportion of "real steps" versus "fake steps" (differentiating between those from zero velocity and those from exiting the boundary) for all data sets, workloads, and seeding volumes. It shows that:

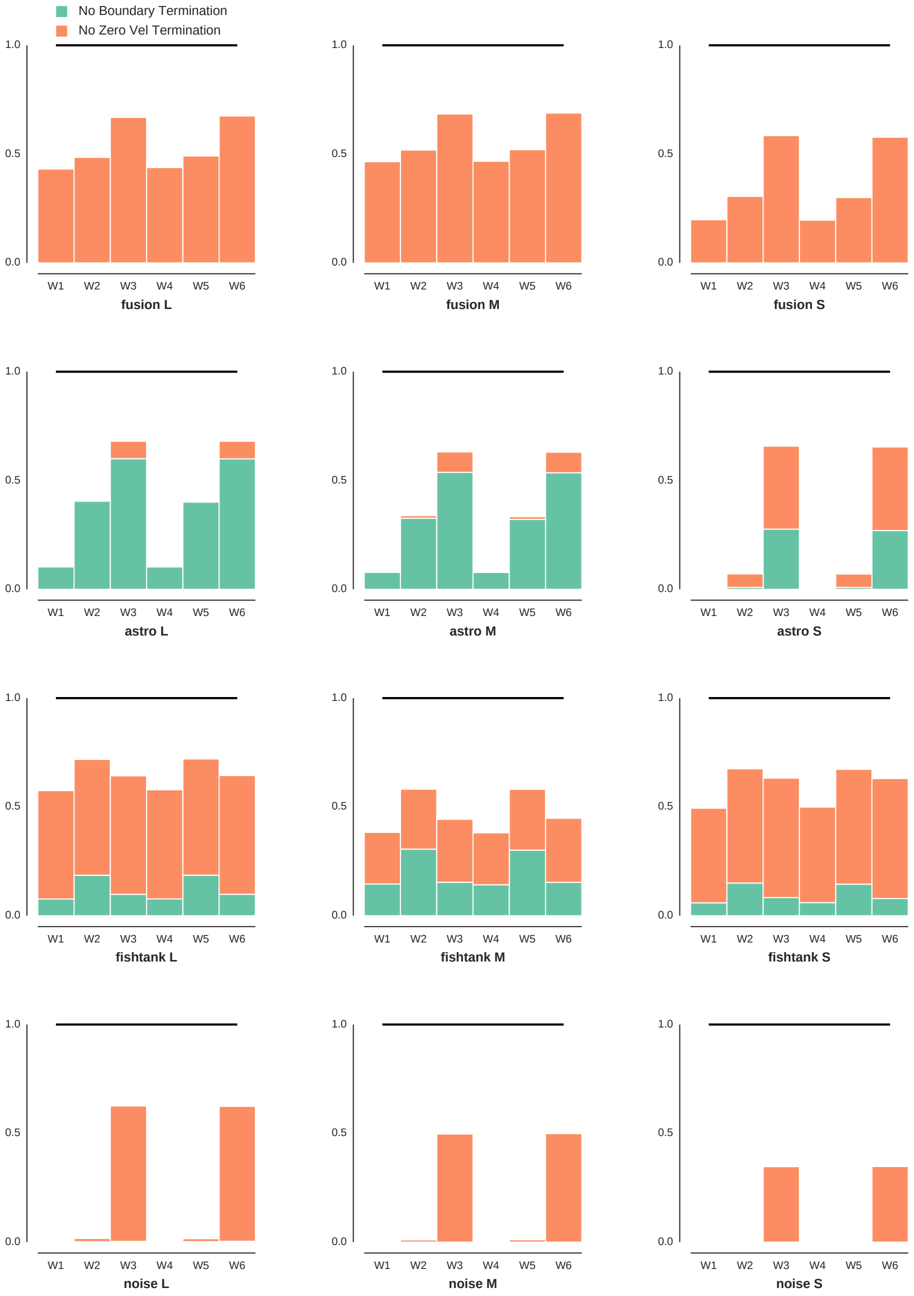


Figure 20. Evaluating the proportion of fake steps contributed by different termination criteria when particles are not terminated at all.

– the Fusion data set is made up of 50% fake steps due to zero velocity (or, rather, that the number of steps taken under normal conditions is 2X less than the workload specifies),

– the Astro data set has a large range of outcomes (from 5% fake steps to 70% fake steps, mostly from exiting the boundary),

– the Fishtank data set always has more than 50% fake steps (from a combination of zero velocity and exiting the boundary), and

– the Noise data set has the least number of fake steps, although workloads with longer durations due ultimately find zero-velocity locations.

These results inform Table 21, which repeats the analysis from Table 20 but incorporates fake steps in the timings. As a result, these experiments have no effects from divergence and slowdowns are solely from caching effects. On the CPU side, Table 21 shows that the Fishtank data set is still slower than the other data sets. In this data set, a given particle will hit the ceiling of an assembly and can move in any direction, and then will recirculate and hit the ceiling again. As a result, this data set is the one that most stresses cache, as each particle can travel through the entire volume. The slowdowns for Fishtank and Fusion are roughly half of those in Table 20, while for Astro and Noise the slowdowns are very similar. In all, these experiments show that data set does affect CPU performance — two data sets appear to be affected by caching and divergence in approximately equal measure, while two other data sets appear to be affected only by caching. On the GPU side, eliminating divergence improved slowdown factors for all four data sets. The biggest effects were for Fishtank, with the GPU4 experiments dropping from a slowdown of 9.58 in Table 20 to 2.62 in Table 21. For this data set, divergence was a much larger cause of slowdown than caching. The effects are smaller for other data sets. For example, the Noise data set on GPU4 improved from 2.35 to 1.90. For this data set, cache performance appears to be a bigger issue than divergence. We are reluctant perform further analysis to quantify the relative effects of each. In particular, the experiments in Table 21 have extra cache benefits from zero-velocity steps, which contributes to the reduction in slowdown.

4.5 Conclusion

The research goals of this chapter were to understand the benefits of using the parallelism of GPUs and multi-core CPUs for particle advection. We identified five research questions that we felt were key to this understanding. In RQ1, our goals were to determine how much speedup a GPU provided over using a serial CPU, and the differences between different GPU architectures. Our study shows that a GPU can provide speedups, but only for certain types of workloads. Over all our tests, GPU speedups were rather modest (2X-5X, depending on architecture). The maximal speedups (6X-25X, depending on architecture) were achieved for the larger workflows consisting of more than 1M steps. Because of memory costs on the GPU, the type of algorithm used has a large impact. The expected speedups across different architectures for the streamline algorithm ranges from 3-9X, while the particle advection algorithm ranges from 4-27X. The takeaway message is that a GPU implementation only makes sense when used on large enough workloads to overcome the costs associated with memory usage and data transfers. Finally, RQ1 also asked about the difference between GPU architectures, and, on the whole, the best GPU in our study was ~4X better than the worst. RQ2 investigated how much speedup could be achieved using multi-core CPUs over using a serial CPU. We found that multi-core parallelism was always useful, and significantly useful if the number of particles was 1000 or more. We also found that the speedup can vary based on the algorithm (streamlines strain memory more) and CPU architecture. As far as comparing CPU architectures, the 104-core Xeon ran fastest for all experiments with more than 100 particles, but the amount of improvement varied. RQ3 investigated if there were clear choices to be made for the community in deciding what type of parallelism to deploy for particle advection tools. For this question, the overall takeaway is that multi-core CPUs tend to be more efficient than GPUs. The costs for data transfer and memory usage are so high that it takes significant amounts of work to overcome. Further, because of the way the algorithms parallelize the work, only large number of particles can engage all of the cores on the GPU. CPUs with a large number of cores are very efficient at advecting particles. RQ4 presented the trends of performance improvement afforded by the advancements in both, GPUs and multi-core CPUs. The findings suggest that the both the architectures are able to benefit form the increasing concurrency of the execution hardware. The difference between the performance of the worse and the best GPUs and CPUs for the largest workloads was 10X. Finally, RQ4 demonstrated that the underlying vector field can have a significant impact on the performance of particle advection. The performance largely depends on the proportion of the total workload that is actually executed, as well as the characteristics (convergence and divergence) of the vector field which affects cache performance.

Our findings are somewhat different than the those by Pugmire et al. [110] who did a smaller study in 2018 comparing GPU and multi-core CPU performance again using VTK-m. We are using different architectures so direct comparison is difficult. That said, their comparisons on “Summit Dev” between Pascal and 20 cores of IBM Power-8 showed more significant speedups on the GPU than our comparisons on Summit between a Volta and 32 cores of IBM Power-9 performance. While none of our experiments match exactly, the most comparable involve our runs of one million particles with one thousand steps versus their runs of ten million particles with 100 steps. They achieved 1.4s with a P100, while we achieved 2.39s. On the CPU side, our run was 4.8s, while their run was 19.9s. The GPU slowndown may be attributable to many factors, but we point to changes in the last four years of the VTK-m source code. During that time, the code has been extended to be "lean and mean" to be functional for a variety of use cases (FTLE, different grid types, electomagnetic fields, etc.) and this has led to more branching, etc., that can slow down GPUs. As a result, our findings should be interpreted as expected performance for a practical, richly-featured implementation.

In summary, we feel this chapter provides information for the visualization community to guide decisions for tool development and deployment for particle advection. It will also help set expectations for algorithm performance on different hardware. This will help focus the efforts of developers to provide efficient solutions for the types of problems they plan to support and hardware that is available. While programming models for GPUs are improving, they are still challenging devices for development and debugging. Realistic expections for the performance on GPUs can be balanced against the development and maintenance cost for the anticipated uses cases of the software. For visualizations tools with large user bases, and use cases that are varied, this work provides valuable information for the development of heuristics that can be used at run-time to make decisions on which hardware to target. Finally, we feel this work has significant implications for distributed memory parallelism and in situ use cases. A distributed memory implementation might need to support very large workloads, but this workload might be spread across a number of nodes. In such a case, there is globally a lot of work to do, but locally only modest amounts of work to do. It would be advantageous to use this study to inform how the node-level particle advection is performed. For in situ processing, there are different options for how algorithms can be run. Is the simulation data already on the GPU? Are there CPU cores idle while the simulation runs? Is the GPU idle while simulation does communication, or switches to using the CPU cores? Is there enough work to warrant a transfer from the CPU to GPU, or vice versa? The best choice will vary based on the answers to these questions, and the findings in this chapter can help inform these decisions.

CHAPTER V

GENERAL PURPOSE FLOW VISUALIZATION FOR THE EXASCALE

This chapter is based on a co-authored work which is in preparation. I am the primary author for this chapter and I will be the first author when the work is submitted. Hank Childs has made editorial suggestions and also made suggestions for overall project direction.

5.1 Introduction

There is a wide diversity of options for flow visualization systems, ranging from velocity field evaluation to solver to parallelism style. Each of these options exists because they have utility in a given setting, i.e., this velocity field evaluation is appropriate for this data, this solver is most efficient for these conditions, or this parallelism is the fastest for this combination of workload and architecture. One option for a visualization flow developer is to identify the specific options for their use case and implement a corresponding system. However, the resulting system may only be useful to a limited set of people, and, stating it explicitly, not applicable to most potential users. As a result, additional systems may get developed, each with their own special purpose. In all, focusing on the specific needs of one use case may result in many different implementations, each coming at high cost.

The premise of this research is that developing a general system capable of satisfying many use cases will obviate the need for many bespoke systems. The value of this approach would primarily be in reduced development time, i.e., one visualization developer implements a general system — likely at a higher cost than a bespoke system — and many future visualization developers can re-use this work. Another value may come in extensibility. Rather than a rigid system that was written for a specific purpose (which can lead to extensibility barriers), a general system would already have extension paths built in, making adding new functionality straightforward.

An important consideration for this thesis is efficient performance on modern supercomputers. This requires two types of parallelism: distributed-memory parallelism and shared-memory parallelism. The first type of parallelism was well explored by Binyahib in her dissertation [15], which considered the efficacy of parallelization-over-data techniques, parallelization-over-particles techniques, and hybrids between the two. Her findings remain very relevant, in particular because the number of nodes on supercomputers is remaining relatively fixed. Instead, the major change is the presence of accelerators and the number of cores available per node. The second type of parallelism was the focus of Chapter 3. In all, the best approaches for both types of parallelism are now well understood, partially from previous work and partially from results earlier in this thesis.

This research for this general system contains two significant elements: First, what are the abstractions? And, second, how can these abstractions be achieved?

The first element, identifying abstractions, is a significant challenge. That said, most of the work for this challenge has already been performed earlier in this dissertation. In particular, Chapter 2 identified the abstractions for the advection process, and the work from Binyahib and Chapter 3 add additional abstractions for parallelization. These abstractions, as well as concrete implementations, can be seen in Figure 21. These choices will be validated in the Results section through demonstrations of diverse flow visualizations, i.e., the ability to support a wide range of use cases provides evidence the choices of abstractions are good ones.

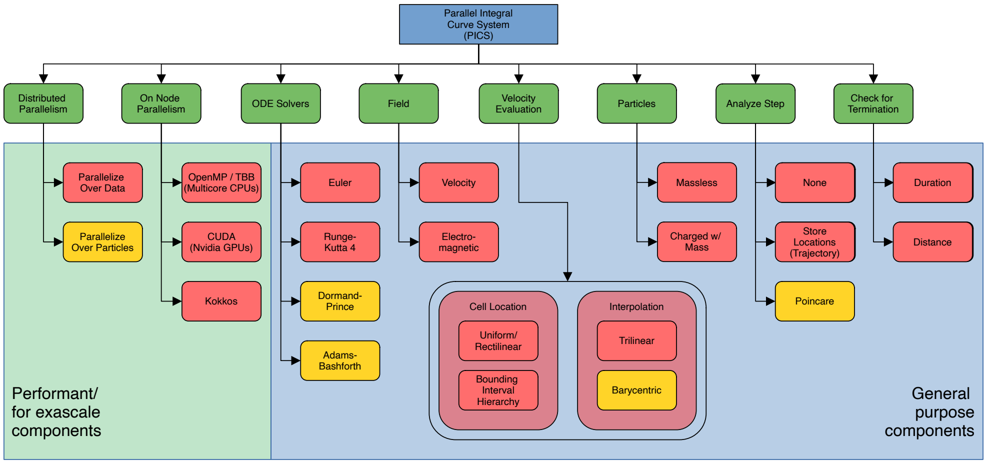


Figure 21. The abstractions and their concrete realizations for our general purpose flow visualization system. The components encapsulated in the green box are the components that make the system exascale ready. The components encapsulated in the blue box are the components that make the system extendible and general purpose. Together these components aim to achieve a goal of "efficiency squared," i.e., both performance efficiency and developer efficiency.

The second challenge is implementing these abstractions. In C++, it is easy to envision a solution using virtual functions. Abstract types provide interfaces via pure virtual functions and concrete types provide implementations for those virtual functions. Each abstraction interacts with the others as appropriate, i.e., a Solver class calls VelocityFieldEvaluator::EvaluateAtPoint as many times as needed to do its work. However, a key goal for this work is to provide a solution that will work on modern supercomputers, and that requires that the code code can run on accelerators. In particular, some GPUs do not support virtual functions. Further, even though some GPUs do support virutal functions, they could still be a poor choice — while the overhead from virtual functions is often small, it can denigrate performance when invoked at a high rate. As a result, a C++-style virtual function approach was not possible for this work. Instead, we focused on template metaprogramming and specifically using traits for specialization. More on this topic can be found in Section 3.

5.2 Related Work

Flow Visualization is supported in widely adapted scientific visualization packages like VisIt [41], Paraview [67], VTK [57], FieldView [1]. Special flow visualization tools are also made explicitly for flow visualization, like OSUFlow [52]. Most of these tools still offer limited support for using advances on multi-core CPUs and accelerator technologies (specifically GPUs) for anything apart from rendering. Also, using these tools for novel analysis becomes challenging as either significant expertise is required to contribute code to them or these tools inherently allow themselves to be extended easily. Hence there is a void in the research for developing the flow visualization tools that satisfy our "efficiency squared" criteria. However, there is plenty of research tackling specific problems using these tools.

5.2.1 Towards Performance Efficiency. Recently, the landscape of high performance computing has been transformed by the advances in accelerator technologies, specifically Graphical Processing Units (GPUs). Previous research on using GPUs for particle advection focuses on interactive visualization since using GPUs for particle advection makes the geometry to produce visualizations readily available on the GPU [78, 27, 26, 24]. Other studies present algorithmic optimizations to components like cell location to efficiently use the available concurrency [54, 118]. These works successfully demonstrated the value of using GPUs for particle advection with performance improvements of up to 80X for interactive visualization and up to 17X using algorithmic improvements for better use of GPU. However, these works demonstrated were explicitly designed to execute on a single hardware platform. The scientific computing community has seen a diversity of accelerator hardware being used to build the latest and greatest supercomputers. The top five current supercomputers use 3 different types of technologies, AMD GPUs, ARM-based CPUs, and Nvidia GPUs. With such a variety of computing landscapes, optimizing software for each of these platforms is not feasible, warranting more research in platform portable technologies. An alternative to remedy this situation is to use “data parallel primitives (DPP),” where most common code patterns are optimized for a specific platform, and developers are encouraged to use these code patterns as building blocks for their algorithms. Using DPPs for particle advection was explored in Chapter III, with findings demonstrating speedups ranging from 0.4X to 2.25X compared to platform-specific implementations [95, 110].

In term of large-scale flow visualization, there is research that abounds. A survey by Zhang et al. [137] summarized a large set of studies that aims to solve flow visualization challenges at scale. At large scales, where work and data need to be distributed among multiple nodes, communication of intermediate results, managing data and I/O, scheduling work, and load balancing become issues of critical importance. Several notable studies presenting novel strategies for improving the efficiency of large-scale particle advection stand out. However, most of these studies have a singular focus, focusing on better communication or better work scheduling, etc. Also, most of these studies primarily use distributed memory parallelism, ignoring the potential performance benefits of using shared memory parallelism. Using shared and distributed memory parallelism together is termed “hybrid parallelism.” With the advances in accelerator technologies and the need to tackle more significant scientific problems, research in efficient hybrid parallelism is becoming increasingly important. Only a few particle advection-based studies demonstrate the use of hybrid parallelism. Key among them were studies presented by Camp et al. and Childs et al. that generated streamlines using hybrid parallelism [39, 32, 33]. More recently, Binyahib et al. demonstrated using VTK-m to develop and test multiple hybrid particle advection algorithms [19, 18, 17].

5.2.2 Towards Developer Efficiency. While no effort has tackled our particular flavor of developer efficiency, many previous visualization efforts have considered this topic. First, VTK-m [95] provides developer efficiency over platforms, i.e., write once, run everywhere. Outside of the visualization community, efforts such as Kokkos [45], RAJA [71], and Thrust [3] do the same (among others). These projects differ from ours because they focus on portable performance. Our effort values portable performance, but it achieves this property via VTK-m and seeks additional developer efficiencies for advection. Second, several visualization systems have provided great flexibility by providing many visualization modules and interoperability capabilities to connect these modules. These projects include VTK [57], AVS [126], and OpenDX [103], among others. Our effort differs because we are considering how to provide developer efficiency at the module level, i.e., instead of providing hundreds of advection-related modules within these systems, we are asking how to develop a single module that can be specialized to realize each of the desired advection modules. Our effort is operating at a different level than these visualization systems.

5.3 Implementation

Figure 21 presents the main abstractions we identify as essential to make the system general and easily extensible. Seven total abstractions are required, of which two are required for efficient execution on heterogeneous supercomputers, and five are required for general flow visualization. This section is divided into four subsections. Section 5.3.1 discusses the necessary VTK-m concepts that help better understand the design of the system. Section 6.2 discusses the abstractions that are required for making a general flow visualization system. Section 5.3.3 discusses the organization of all the abstractions inside and the instantiation of a VTK-m filter. Section 5.3.4 discusses the process of specifying a new particle advection-based filter in VTK-m.

5.3.1 Terminology. In HPC terminology, to better distinguish between the CPU and the accelerator being used, the CPU is termed as a "host," and the accelerator device is called a "device". This also extends to refer to the memory for each device; The main memory for the CPU is termed "host memory", and the dedicated memory for the the accelerator is termed "device memory." Users have to explicitly control copying the memory from the host to the device before processing and later from the device to the host post-processing. VTK-m allows users to specify complex objects composed of multiple arrays. However, users must specify how these objects manage host and device memories. To represent these objects, VTK-m terms the objects that reside in host memory as "control objects" and their counterparts on the accelerators as "execution objects." For each of the abstractions introduced in our flow visualization system later in this section, there are specific requirements users need to follow while writing their custom objects such that they work with the infrastructure defined by our system. In each of these abstractions, we'll define what the control object requires and what the execution objects require. For the most part, control objects serve just as containers for arrays on the host side, and if they serve any other particular role, it will be discussed in the relevant section. For a more detailed discussion of VTK-m's control and execution objects, we recommend referring to the VTK-m user's guide.

5.3.2 Achieving Developer Efficiency and Generality.

Integration-based flow visualization is used in multiple scientific domains, e.g., aerodynamics, fusion, combustion, etc. These scientific domains have different needs for visualization and analysis of the flow being represented. However, these flow visualization algorithms share many of the same underpinnings, allowing for the potential for generalizations and abstractions. In theory, these abstractions should enable users/developers to quickly compose new algorithms by specifying custom components that can work with each other to achieve analysis needs. Towards that end, our system provides abstractions of the following components of an integration-based flow visualization system.

5.3.2.1 Particle Abstraction. 'Particle' abstraction allows users to specify special properties that are required to advect or analyze the particle. Typically study of fields like aerodynamics requires massless particles for analysis and requires no unique properties. However, domains like fusion require particles to have momentum and charge. Since particles only represent a container for their properties which we assume to mostly be single entities, there is no separation between control and execution objects.

5.3.2.2 Evaluator Abstraction. 'Evaluator' abstraction allows users to specify how the field will be resolved at a certain location, depending on how the field was discretized for the simulation. Based on popular mesh types supported by VTK-m, out implementations support the evaluation of fields in uniform, rectilinear, or unstructured meshes. The evaluator acts as a means to access the data from the vector fields using a cell locator that works over the input mesh.

Hence, on the control object, it maintains the control objects of the field and the cell locator. On the execution object, it maintains the execution objects of the field and the cell locator. It uses the following methods on the execution object to query and return the relevant field information.

```
ErrorCode Evaluate(const Point& point,
                   const float& time,
                   FieldOutputType& out)
```

In the above code listing, ***point*** is the location where the evaluation needs to happen, ***time*** is used while dealing with temporal data to calculate the velocity at a specific time, and ***out*** represents the value to be returned.

5.3.2.3 Field Abstraction.

'Field' abstraction allows users to easily specify the vector fields representing the flow. However, the specification of a flow field can be complicated. E.g., vector fields for aerodynamics that are discretized into a velocity field are sufficient for flow visualization, while the study of fusion and electromagnetics requires an electric field and a magnetic field. The control side object for the field only contains the arrays representing the flow. The execution object must support one of the following three methods to return the necessary data. These methods are used by the 'Evaluator' to return the correct velocity information or to return all the information needed to calculate the velocity.

First, if the field is zonal, the following method needs to be supported:

```
void GetValue(const Id cellId,
        OutputType& value) const
```

In the above code listing, ***cellId*** refers to the identity of the cell identified by the cell locator for the query location, and ***value*** refers to the relevant information to be returned.

Second, if the field is zonal, the following method needs to be supported:

```
void GetValue(const Vec<Id, 8>& indices,
         const Id numVertices,
         const Vec3f& parametric,
         const UInt8 cellShape,
         OutputType& value) const
```

In the above code listing, ***indices*** refers to the indices of the vertices that make up the containing cell for the query location, ***numVertices*** refers to the number of vertices, ***parametric*** refers to the parametric coordinates of the query location within the cell, and ***cellShape*** refers to the cell's shape so that correct interpolation schemes can be used.

Finally, if a more complicated strategy is needed for field evaluation, the entire process is delegated to the field using the following method:

```
template <typename Point,
       typename Locator,
       typename Helper>
void GetValue(const Point& point,
         const float& time,
         OutputType& out,
         const Locator& locator,
         const Helper& helper) const
```

Here, the first three parameters are similar to the Evaluator objects. In addition to them, ***locator*** refers to the object of cell locator that might be needed to perform query cells, and the ***helper*** refers to an internal object used to query more information like indices of the vertices representing the cell, querying the cell shape, etc.

5.3.2.4 Analysis Abstraction. 'Analysis' abstraction allows users to specify custom analysis based on successive positions of the particle. Users can also choose a wide variety of analysis tasks to perform during particle advection, e.g., storing successive locations of a particle for streamlines and storing intersections with a set of planes for Poincaré plots. Typically, the analysis class is where users need to specialize many more things on the control side to guide the process of output construction. Three main methods are used to accomplish the task of analysis on the control side.

First, users need to specify if the analysis requires developers to handle the initial state of the particle.

```
void InitializeAnalysis(
 const Array<ParticleType>& array)
```

In the code listing, ***array*** refers to the initial state of the particles that need to be advected.

Second, users need to specify if the analysis requires developers to handle the final state of the particle.

```
void FinalizeAnalysis(
 const Array<ParticleType>& array)
```

In the code listing, ***array*** refers to the final state of the particles that have been advected.

Finally, users need to specify how to generate the output for the analysis.

```
void GetOutput(DataSet& data)
```

In the code listing, ***data*** refers to the data set that will contain the output of the current analysis instance. Users will call this method once the entire algorithm has finished execution.

On the execution side, however, users only need to specify what must be done between two successive particle states.

```
void Analyze(const Id index,
        const Particle& oldParticle,
        Particle& newParticle)
```

In the code listing, ***index*** marks the location of the particle relative to other particles at the beginning of the analysis, ***oldParticle*** refers to the previous state of the particle, and ***newParticle*** refers to the new state of the particle. The 'Analyze' method is called for the particle when it's completed a new step.

5.3.2.5 Termination Criteria Abstraction.

'Termination Criteria' abstraction allows users to specify conditions where particles should stop advection, in other words, terminate. Based on the analysis needs, users can also choose different termination criteria for particles, e.g., for generating flow maps and streamlines, users may use duration as termination criteria. In contrast, for generating Poincaré plots, users may use the number of punctures as termination criteria. The implementation of the termination abstraction is quite straightforward.

```
void CheckTermination(Particle& particle)
```

In the code listing, ***particle*** represents the particle whose termination is being queried based on the properties (time, position, etc.) set on it.

5.3.2.6 Solver Abstraction.

'Solver' abstraction allows users to specify custom solvers for their analysis needs. Different scientific domains also require solvers with varying accuracy. E.g., in electromagnetic simulations, one might use the Boris solver. In contrast, one might use the RK4 or similar higher-order solver for aerodynamics. These solvers also present different performance-accuracy trade-offs, since low-order, low-accuracy solvers also require fewer velocity evaluations than higher-order evaluations. The solver has the responsibility to solve for the particle's velocity.

```
ErrorCode Solve(const Particle& particle,
          const float& deltaT,
          Vec3f& velocity)
```

In the code listing, ***particle*** refers to the particle being advected, ***deltaT*** refers to the step length for the solver, and ***velocity*** is the solved velocity for the particle with which it will be displaced.

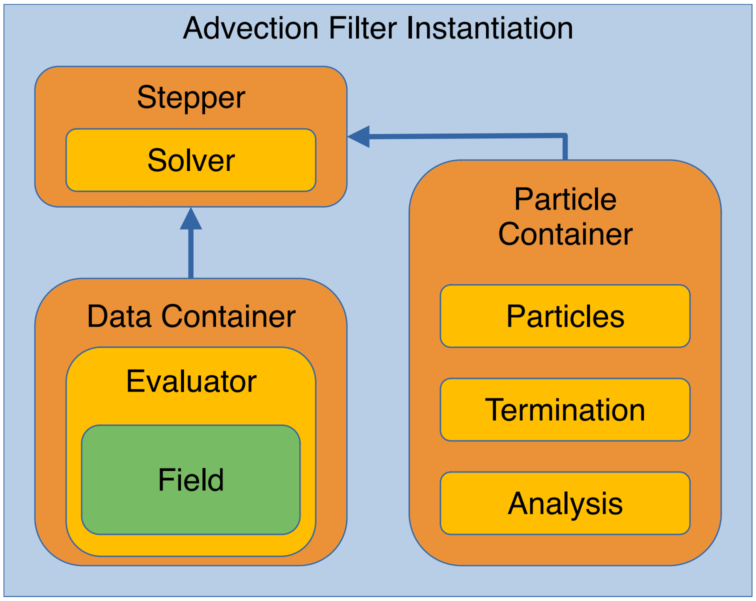


Figure 22. The organization of components using the abstractions discussed earlier for a particle advection-based flow visualization filter. The outer blue box represents a flow filter, and the small yellow boxes represent all the components required by the filter. The orange boxes encapsulate the yellow boxes that represent the components shared throughout the filter execution, while the yellow boxes represent components related to the particles being advected at the moment.

5.3.3 Organization and Interaction. Figure 22 describes the organization of the components in a VTK-m filter instantiation. Based on the user-provided inputs, the VTK-m system creates essential objects for the current context of particle advection. "Particle Container" contains all the objects that modify properties of the particle being advected. First, it stores all the particles being advected as an array represented by "Particles." Second, it guides the lifecycle of all particles by querying the "Termination" object if a particle should be terminated. Finally, it is responsible for feeding the old and current state of the particles to the "Analysis" class so that the output for the algorithm can be generated.

"Data Container" contains all the means to access the data required to calculate the particle's velocity. It maintains an object of the "Evaluator," which maintains all the necessary means to query the data arrays maintained in the "Field." For that, it builds the cell locator based on the coordinates and cells of the current data and some other utilities to help interpolate quantities. "Stepper" is the main driving class for the entire advection process and maintains the instance of the "Solver." It operates over each particle represented in the particle container and advects it one step and a time.

5.3.4 Specifying a VTK-m Flow Filter. VTK-m achieves the generality for flow visualization and analysis by the virtue of "trait-based template programming." Trait-based template programming allows for modification of the behavior of the trait class based on specialization. The process involves specifying the specialization of a "FlowTraits" class which contains the information of all required realizations of abstractions for the filter being defined.

```
template <typename Derived>
```

```
struct FlowTraits;
```

Hence, writing new filters in VTK-m is as easy as writing a new ***FlowTraits*** class for the filter. The trait class defines four things: a flow visualization filter depends on the particle type, the termination criteria, the analysis performed, and the type of vector field used. E.g., the streamline filter in VTK-m is implemented by specifying the following trait class.

```
template<>
struct FlowTraits<Streamline>
{
 using ParticleType = Particle;
 using Termination = NormalTermination;
 using Analysis    = StreamlineAnalysis;
 using Field       = VelocityField;
}
```

Here, the trait class is specialized for the “Streamline” filter, which uses a simple particle and requires normal termination while it does the streamline analysis. The process for simple streamline generation requires a simple velocity field. Section 5.4 describes these specializations for three scientific use cases.

5.4 Results

To demonstrate the efficacy of our system, we ran three campaigns and an additional campaign to evaluate the system’s performance. The first two campaigns were: the WarpX simulation code (Section 5.4.1) and the XGC simulation code (Section 5.4.2). These two campaigns were selected for this study for two reasons. First, they represent real-world exascale use cases with simulation code teams that will run on the exascale computer when it comes online. Second, their requirements have key differences compared to typical flow visualization, so their inclusion demonstrates the system's flexibility. For the third campaign, we designed a campaign to maximize diversity compared to the first two. Since the first two were very non-traditional, the third campaign looked like a traditional flow visualization use case (RK4 on a velocity field defined on a rectilinear grid). It used the CloverLeaf simulation code and is described in Section 5.4.3. Finally, the performance evaluation for the implemented system was performed against a widely used tool for large-scale visualization, VisIt [40], and is discussed in Section 5.4.4 The options for each campaign can be found in Table 22.

Table 22. Abstractions and their realizations for the three scientific use cases and the performance study considered to demonstrate the efficacy of the general purpose particle advection system.

	Realization			
Abstraction	Streamlines for WarpX	Poincaré for XGC	FTLE for CloverLeaf3D	Performance study for CloverLeaf3D
ODE solver	Euler Solver	RK4 Solver	RK4 Solver	RK4 Solver
Field	Electromagnetic	Specialized Electromagnetic	Velocity	Velocity
Location	Rectilinear	Unstructured	Structured	Structured
Interpolation	Trilinear	Barycentric	Trilinear	Trilinear
Particles	Charged w/ mass	Massless Particle	Massless Particle	Massless Particle
Analysis	Streamlines	Poincaré	FTLE	Streamines
Termination	Duration	Number of Punctures	Duration	Duration
On Node Parallelism	CUDA	Kokkos + HIP	OpenMP	OpenMP
Distributed Parallelism	over Data	over Data	over Data	over Data

5.4.1 Streamlines for WarpX. Our first scientific use case focuses on rendering streamlines for WarpX. WarpX is an advanced electromagnetic Particle-In-Cell code and can be used in many domains of laser-plasma science, plasma physics, accelerator physics, and beyond [129, 128]. The US DOE Exascale Computing Project primarily funds it, and WarpX contributors are LBNL, LLNL, CEA-LIDYL, SLAC, DESY, CERN, and Modern Electron. We acknowledge all WarpX contributors. The data from WarpX was rectilinear, divided into 64 blocks of equal dimensions of 256 × 256 × 512. The experiments with WarpX were conducted on the Crusher supercomputer hosted at the Oak Ridge National

Laboratory, using AMD MI250X GPUs as accelerators. The experiments were conducted using 8 nodes and 64 tasks, with each node running 8 tasks and each task running on 1 GPU using Kokkos with the HIP backend. The abstractions that were used for this use case are described in Table 22.

The following code listing shows the required specialization of the trait class for specifying a filter that can generate streamlines for WarpX.

```
template<>
struct FlowTraits<WarpXStreamline>
{
 using ParticleType = ChargedParticle;
 using Termination = NormalTermination;
 using Analysis    = StreamlineAnalysis;
 using Field       = ElectroMagneticField;
}
```

Finally, Figure 23 presents visualizations of the streamlines generated for the WarpX use case. The streamlines represent the movement of electrons in a laser wakefield.

5.4.2 Poincaré for XGC.

XGC is a gyrokinetic particle-in-cell code that specializes in the simulation of the edge region of magnetically confined thermonuclear fusion plasma [62, 63, 79]. The data from XGC was unstructured, with a total of X triangular cells representing a slice of the tokamak. The experiments with XGC were conducted on the Summit supercomputer hosted at the Oak Ridge National Laboratory, using an Nvidia V100 GPU as an accelerator. The experiments were conducted using 1 node and 1 task using the CUDA backend for shared memory parallelism using GPUs. The abstractions that were used for this use case are described in Table 22

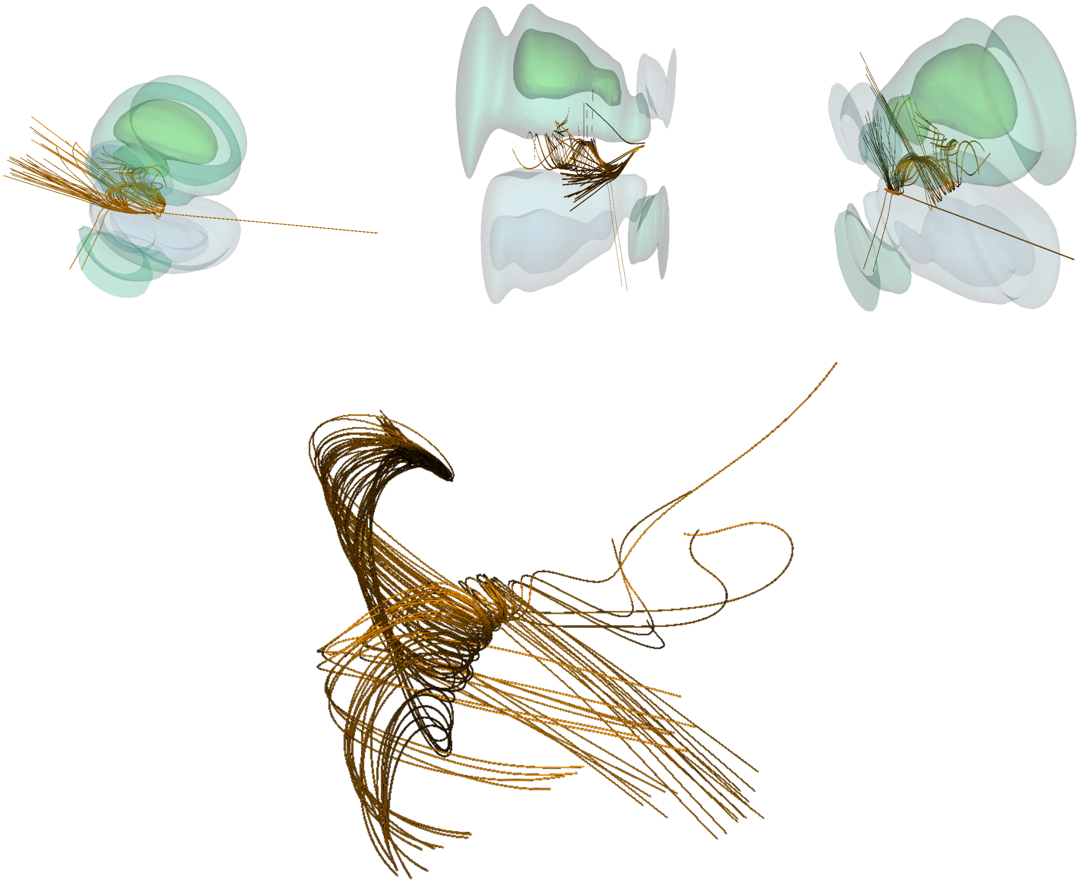


Figure 23. Streamline visualizations for WarpX data. The Y-component of the electric field iso-surfaces colors the green blobs in the top subfigures. The bottom subfigures plot the only streamlines, with the top-left being the initial location of the electrons. A total of 50 streamlines, representing the trajectories of 50 electrons, are shown in the bottom subfigure.

The following code listing shows the required specialization of the trait class for specifying a filter that can generate Poincaré plots for XGC.

```
template<>
struct FlowTraits<XGCPoincare>
{
 using ParticleType = Particle;
 using Termination  = PoincareTermination;
 using Analysis     = PoincareAnalysis;
```

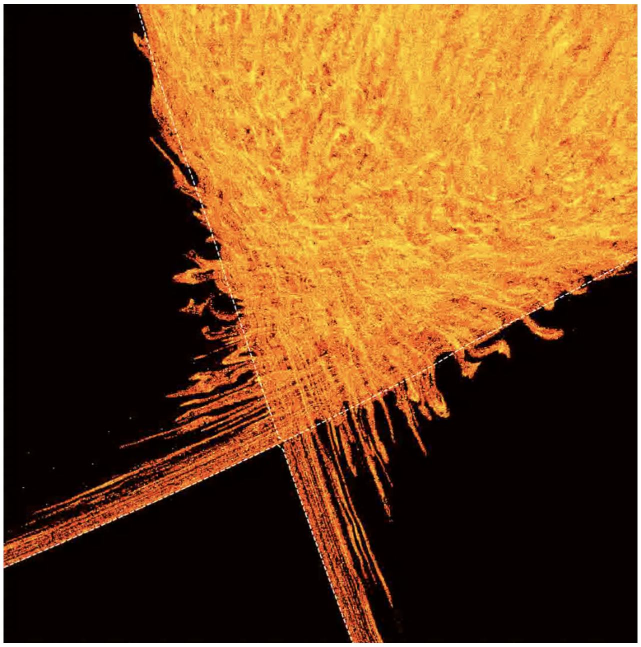


Figure 24. Poincaré (puncture) plots generated for the XGC app. This image is courtesy of David Pugmire and contains additional post-processing after the Poincaré punctures are generated.

```
  using Field      = XGCField;
}
```

Finally, Figure 24 presents the visualizations representing the Poincarë plots for the XGC use case. The visualization represents the turbulent homoclinic tangles.

5.4.3 FTLE for Cloverleaf. CloverLeaf is a mini-app that solves the compressible Euler equations on a Cartesian grid using an explicit, second-order accurate method. Each cell stores three values: energy, density, and pressure. The data from CloverLeaf3D was structured with the dimensions 261 × 133 × 261. The experiments with CloverLeaf3D were conducted on the Summit supercomputer hosted at the Oak Ridge National Laboratory, using IBM Power9 multi-core CPUs. The experiments user conducted using 1 nodes and 1 task using the OpenMP

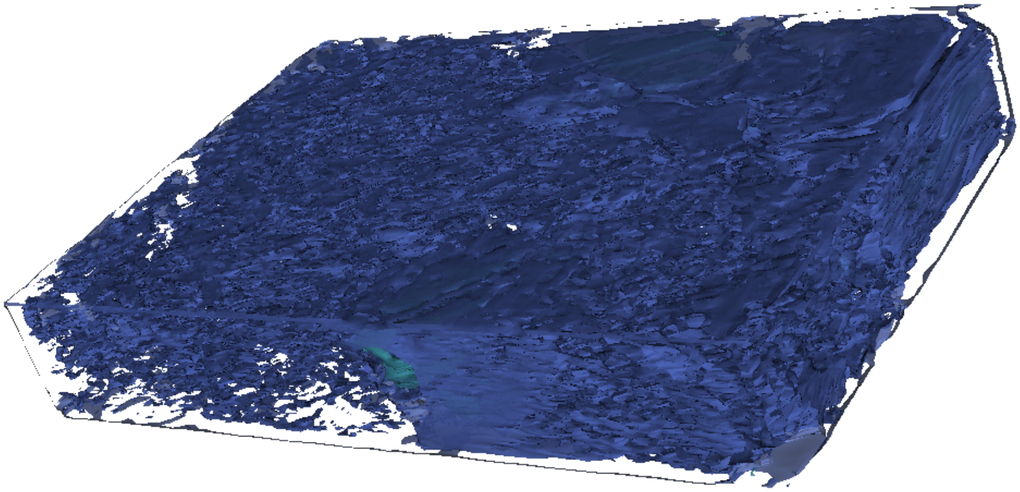


Figure 25. Iso-surfaces extracted using 5 different values of the FTLE field generated using CloverLeaf3D's velocity field.

backend for shared memory parallelism using multi-core CPUs. A total of 1 Million seeds, going for 100 seeds each, were used to generate the FTLE field. The abstractions that were used for this use case are described in Table 22

The following code listing shows the required specialization of the trait class for specifying a filter that can generate FTLE fields for a general case.

```
template<>
struct FlowTraits<FTLE>
{
 using ParticleType = Particle;
 using Termination  = NormalTermination;
 using Analysis     = FTLEAnalysis;
 using Field        = VelocityField;
}
```

Finally, Figure 25 presents the visualization of the iso-surfaces extracted from the FTLE field generated using VTK-m. The contours represent the areas that share similar convergence or divergence in the velocity field.

5.4.4 Performance Evaluations using Cloverleaf.

For the performance tests with CloverLeaf, the original data was resampled into 27 blocks of rectilinear data with with the dimensions 384 × 384 × 384. The experiments with CloverLeaf3D were conducted on the Summit supercomputer hosted at the Oak Ridge National Laboratory, using IBM Power9 multi-core CPUs, using 27 nodes, with each task using a maximum of 42 CPU cores. A total of 4 experiments with a varying number of seeds were considered to compare the VTK-m implementation against VisIt. The abstractions that were used for this use case are described in Table 22

The following code listing shows the required specialization of the trait class for specifying a filter that can generate FTLE fields for a general case.

```
template<>
struct FlowTraits<FTLE>
{
 using ParticleType = Particle;
 using Termination = NormalTermination;
 using Analysis    = StreamlineAnalysis;
 using Field      = VelocityField;
}
```

Figure 26 presents the visualization of streamlines obtained from the CloverLeaf simulation for reference.

Table 23 presents the results of our performance experiments with our system and VisIt. The results show that VTK-m is able to outperform Visit in the case of generating streamlines even when run in serial mode. VisIt is a fully featured visualization tool, and we believe it incurs a performance penalty in the way it treats data to make consequent operations (like rendering) more

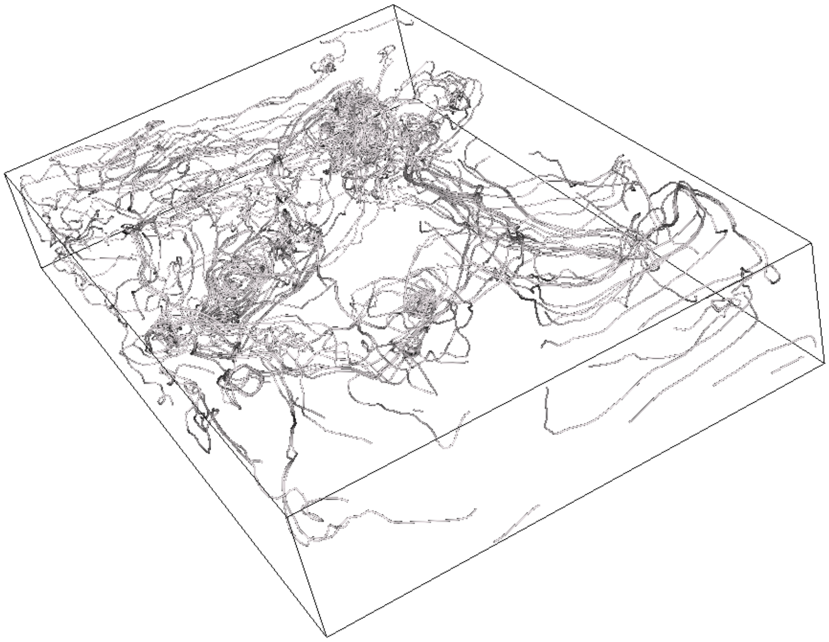


Figure 26. Streamlines generated using the CloverLeaf data using 100 seeds being advected for 10000 steps each.

Table 23. Comparison of performance against VisIt for large-scale experiments of streamline visualizations using CloverLeaf data. The number in parenthesis following VTK-m represents the number of CPU threads used by VTK-m. All timing are measured in seconds.

# of Particles	VisIt	VTK-m (1)	VTK-m (42)	VTK-m (1) vs. VisIt	VTK-m (42) vs. VisIt	Avg. Seeds/ Proc
100	0.42	0.09	0.08	4.50	4.95	3
500	1.20	0.52	0.35	2.29	3.35	18
1000	2.25	1.24	0.86	1.81	2.60	37
5000	10.55	6.37	3.94	1.65	2.67	185

convenient. These penalties are significant when the amount of computation is low. Hence, even in serial execution, VTK-m performs 4.5X faster when generating 100 streamlines, but the advantage is reduced to 1.65X when generating 5000 streamlines. While using parallel execution, VTK-m can afford better performance, but not in proportion to the 42 CPU cores used by the OpenMP backend. This is because the number of particles per task is very low to mitigate the overhead incurred by OpenMP in managing work in concurrent threads. On average, the smallest workload had only 3 particles per task, and the largest workload had only 185 particles. More work is needed to make efficient use of available parallelism. We planned on performing more experiments with a larger number of particles but were limited by the availability of the cluster (Summit) used for the experiments at this time. We reserve a thorough performance evaluation for the future.

5.5 Conclusion

This chapter is both short in length and the culmination of the dissertation. It is short since it benefits from the works before it, in particular the identification of abstractions in Chapter 2 and the data-parallel approach in Chapter 3. The use of template meta-programming and traits for specialization allowed us to combine our abstractions on GPUs, which was necessary for our goal of achieving an exascale capability. Finally, we believe the demonstrations with XGC and WarpX are noteworthy — these are important simulation codes for the US Department of Energy's Exascale Computing Program and will be among the first to run on the exascale machine when it comes online. Further, they represent exactly the sort of diversity that has led to bespoke solutions in the past, and we were pleased that our system could accommodate them. This work does suggest future work; this is discussed in the final chapter.

CHAPTER VI

CONCLUSION AND FUTURE WORK

I was the primary author for this chapter, and Hank Childs provided editorial suggestions.

The scientific community is at the gates of the exascale. The availability of such powerful supercomputers is going to make it easier to tackle larger scientific problems, but while doing so, will also create new challenges in terms of processing and interpreting these results. Simulation codes will be able to generate data at much faster rates and much higher volumes than the ability to store the data to disk, hence, requiring scientists and developer to research ways to analyze and reduce data as it is being generated, a process referred to as “in situ analysis.” In situ analysis requires the data analysis and data reduction algorithms to run besides the simulation code and hence need to be efficient so that they do not encumber the simulation time and resources. Hence, understanding the nature and performance characteristics of the analysis algorithms becomes critically important.

This dissertation focused on analysis algorithms that are often used to analyze data from fluid simulations. In particular, it focused on algorithms that use “particle advection,” to drive flow visualization and analysis. The question posed for this dissertation study was: “What flow visualization system designs will enable both efficiency on exascale systems and be capable of supporting diverse analysis needs?” The question was further broken down into two different objectives:

- What methods and approaches will enable efficient performance on exascale machines?
- What system design can both address diverse analysis needs while also delivering performance on exascale machines?

This chapter describes the takeaways from this dissertation in context of the two question. First, 6.1 describes takeaways from our efforts to answer the first question. Second, 6.2 described takeaways from our efforts to answer the second question.

6.1 Efficient Approaches for Particle Advection

This section answers the question "What methods and approaches will enable efficient performance on exascale machines?" The research conducted towards this question was presented in Chapters II, III, and IV, each of which covered the problem of particle advection from different perspectives.

First, Chapter II surveyed the field of flow visualization works that used particle advection and studies the different kinds of optimizations to improve the efficiency of the algorithms. These optimizations were separated into two parts: "Algorithmic," which covered the optimizations that are possible for the individual components involved in calculating the particle's next position, and "Hardware Efficiency," which covered using different types of shared and distributed memory parallelisms. This was a first-of-a-kind survey to document all possible optimizations and to establish the expected range of benefits by incorporating each of these optimizations. This chapter also introduced a novel analytical cost model for particle advection workloads and tried to validate it using empirical results. The objective of the cost model was to equip users with a workflow for choosing optimizations based on their budget for executing the workload.

Second, Chapter III introduced a particle advection framework that uses "Data Parallel Primitives (DPPs)," through the VTK-m library to make the framework portable. Using DPPs allows the algorithm to execute efficiently on a wide variety of shared memory parallel environments. The VTK-m implementation demonstrated efficient execution and good scalability, while outperforming the hand tuned reference implementations in most experiments. Overall this chapter helped in establishing the efficacy of the proposed platform portable design for particle advection, which was further improved to support additional types of flow visualizations and analysis.

Finally, Chapter IV considered multiple variables: the number of particles, the duration of advection, the algorithm (advection or streamline), the execution device (4 CPUs or 4 GPUs), and memory transfers for GPUs (with and without) to analyze which factors dominate the performance of particle advection workloads. The study also studied the impact of the dataset on the performance of particle advection. The findings from this chapter enable users to make better decisions about executing their workloads on a system where parallelism is available. In particular, the findings suggest GPUs are often poor means of parallelism when memory needs to be transferred back and forth between the CPU and the GPU and when there's not enough work to be done to offset these memory costs. This already narrows the usability of GPUs for particle advection to workloads which have a large number of particles, however, at such high workloads the limited dedicated memory with the GPU becomes restrictive.

In all, the contributions from Chapter II provide a high-level understanding of costs for various approaches. That said, this understanding does provide sufficient insight into the issue of parallelization, which is critical for exascale machines. Chapter III then provided the "how" for algorithmic design, namely to use data-parallel primitives to achieve portable performance. Finally, Chapter IV informed the specific outcomes for different hardware and workloads, complementing the missing information from Chapter 2 to inform the overall space.

6.1.1 Future Work.

The work discussed in each of the above mentioned Chapters also identifies scope for more research for the future.

Chapter II introduced the an analytical cost model, however, the cost model was not used to make predictions about the performance of real world applications. The empirical validation performed was also very trivial. As future work, this cost model can be improved to be a hybrid approach, combining the analytical and empirical approaches of cost modeling to better predict the execution times of a flow visualization algorithm. The evaluation also focused on particle tracing alone, and can be extended to reasoning about the performance of additional use cases, e.g. streamlines, Poincaré analysis, etc.

Both, Chapter III and IV focus on using VTK-m to evaluate the efficacy of using DDPs and to identify the dominant factors for particle advection performance respectively. These studies can be further improved by considering the "roofline model," to measure the achieved performance or efficiency by the algorithms. The roofline model can help establish a range of efficiency for the VTK-m implementation of particle advection, and also can be a quantitative measure for the dominant factors of particle advection.

Finally, Chapter IV makes interesting observations about the performance of particle advection on GPUs and multi-core CPUs, in turn establishing when each of these execution devices are helpful over the other. The behaviors observed were consistent on multiple generations of CPUs and CPUs. Using these observations, it is possible to design a system of particle advection that is able to make runtime decisions about which execution device to choose, also termed as "device targeting." Such a system will be very helpful in large scale experiments involving distributed memory parallelism, where GPUs do not consistently have enough work and using them might affect performance in a negative way.

6.2 Achieving Generality for Flow Visualization

The demonstrations from Chapter 5 show a system design for achieving generality and still maintaining performance. A key approach was embracing template metaprogramming with specialization through traits. It is my belief that this system is without comparator. The only close comparator is the one in VisIt, but its infrastructure cannot support accelerators for GPUs. Further, the WarpX and XGC campaigns show this code can handle a diversity beyond any previous system — another differentiating point from VisIt.

Going forward, there are several more concrete types that would make the system more useful. More of the distributed-memory parallelism techniques from Binyahib should be added, creating more opportunities. That said, this library will primarily be used for in situ processing, making the existing parallelize-over-data far and away the most useful (since it does not require moving data). Further, additional useful vortex and feature detection analysis like Q-Criterion, additional higher order solvers like Dormand Prince and Adams Bashforth, will make the system desirable to an even broader scientific community.

APPENDIX A

COST MODEL VALIDATION

Chapter II introduced a nascent cost model for particle advection, however, it did not demonstrate its usefulness in determining the execution times for a particle advection workload. The ability to determine the execution times for a workload will enable the realization of the optimization workflow presented in the conclusion of this chapter. To that end, this appendix evaluates our cost model for particle advection performance in three parts. First, it considers the actual costs for terms in the cost model, measured in floating point operations. Second, it considers a notional example of how the cost model can be used to predict the number of floating point operations. Finally, it provides an evaluation of the cost model's accuracy on various workloads, and also provides an approach for converting the output of the cost model to execution time.

A.1 Evaluating Cost Model Terms in Floating-Point Operations

Equation 2.6 does not specify the unit of measurement for each term and for the overall cost. While time would be a common choice, we choose our unit of measurement to be number of floating-point operations. We make this choice since floating-point operations are consistent over architecture and caching effects, and, further, are easy to quantify (either by studying code or through profilers). Finally, looking ahead to validation, counting floating-point operations does track closely with actual execution.

Table A.1 presents the floating-point costs for the terms in Equation 2.6 for a variety of instantiations: RK4 and Euler solvers and three commonly used types of meshes. The terms is this table were determined by studying code and counting floating-point operations. As noted in the table caption, some operations vary

Table A.1. Analytical cost calculation for particle advection. The costs in the table are by reference of a 50^3 grid. Hence, for the next two equations, $d = 50$. Rectilinear location costs : $3 \times log(d)$. Unstructured location costs : $log(d^3)\ times 10 + 748$. The costs for unstructured grid are highlighted in red as these are estimates based on a tree structure. This study assumes 10 FLOPs for checking each level of the tree. An additional 748 FLOPs are required to check if the point indeed belongs inside the identified containing cell which is estimated using the Newton's method. Each iteration of the Newton's method requires 374 FLOPs (code reviewed from VisIt), and based on our experimental validation each check required 2 iterations to converge.

Solver	Data set type	*solve*	*locate*	*interp*	*terminate*	Total
Euler	Uniform	6	15	15	5	41
	Rectilinear	6	17	15	5	43
	Unstructured	6	918	35	5	964
RK4	Uniform	37	15	15	5	162
	Rectilinear	37	17	15	5	170
	Unstructured	37	918	35	5	3854

based on mesh size and the table entries correspond to a typical size. Finally, cell location with unstructured meshes incorporate Newton's method, and we performed experiments to find the average number of iterations to converge (2 iterations).

A.2 Notional Usage of the Cost Model

This section demonstrates cost estimation for a hypothetical workload based on Equation 2.6 and Table A.1. The hypothetical workloads consists of a flow visualization algorithm advancing a million particles in a 3D uniform grid for a maximum of 1000 steps using a RK4 solver. Then, the cost of each component can be estimated as follows:

Locate: The locate operation in a uniform grid uses 15 FLOP to find the cell and the particle's location within the cell for interpolation.

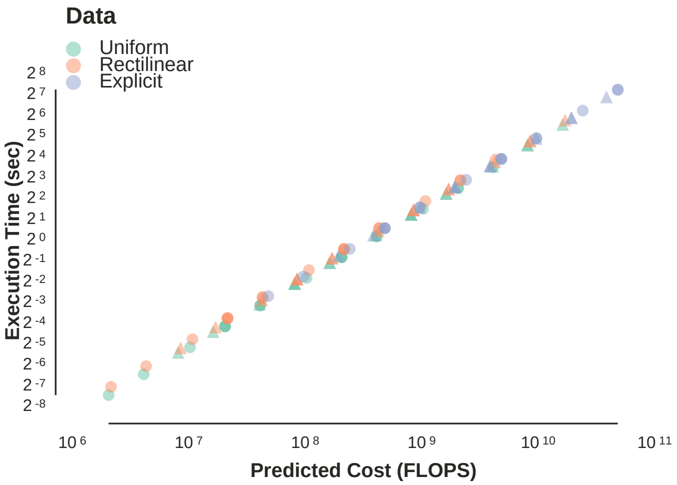


Figure A.1. Comparing analytical cost and actual execution cost for particle advection. The first four columns describe the workload. The X axis represents the estimated number of FLOPS for a workload using information from Table A.1 and our cost formula (Equation 2.6). The Y axis represents the execution time of running this workload on a single core using VTK-m. All experimental points being collinear would build confidence that an analytic approach can be used to estimate runtimes.

Interpolate: The interpolate operation in a uniform grid requires trilinear interpolation which uses 15 FLOP to evaluate the velocity at the given location.

Solve: The solve operation for the RK4 integration scheme uses 37 FLOP to calculate the determine the next position of the particle.

Analyze: The cost of analysis of the step varies based on the visualization technique being used. In case only deals with advancing particles and hence the analysis cost is 0.

Terminate: The terminate operation requires 5 FLOP to determine if the particle is outside the spatio-temporal bounds or if the particles completes the maximum number of steps.

These costs can be substituted in Equation 2.6 to get the final cost of a single step of a particle in the presented situation.

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

A.3 Empirical Validation

This section performs experimental validation of our cost model (Equation 2.6) incorporating our measurements for the number of floating-point operations per term (Table A.1). It presents a set of experiments performed where the calculated cost is translated into an execution time for a workload and them compared against its actual execution time. These experiments were performed on an Intel Xeon E5-1650 CPU with a clock rate of 3.80 GHz using a particle advection implementation from the VTK-m visualization library [95] with a single CPU core. The data used for the experiments was of the resolution 50 × 50 × 50, which matches the assumptions in Table A.1.

The results of the experiments are presented in Figure A.1. This figure shows a very strong fit between the predicted cost in FLOPS and the actual execution time; statistical analysis shows a correlation coefficient of $\boldsymbol{X}$, and a best fit line of $\boldsymbol{Y}$=$\boldsymbol{mx}$+$\boldsymbol{b}$. In effect, the best fit line provides the actual time prediction. For example, a workload with 10^8 FLOPS would take $m \times 10^8 + b$ seconds.

In the context of our workflow, a visualization application developer may choose to do more work. First, they could run several experiments on their intended architecture to calculate their own best fit line. Second, they could recalculate Table A.1 using their own implementation and/or anticipated data sizes. That said, performing such additional work is likely unnecessary. Our model and workflow are intended to infer coarse trends, and repeating our analysis with new implementations, architectures, or data sets would likely not yield a significantly different cost model.

APPENDIX B

GPU AND CPU PERFORMANCE ON DIFFERENT DATA SETS

Chapter IV evaluated the impact of data sets on the performance of particle advection. This appendix provided additional analysis to understand this impact better by expanding the data summarized in Table 20 and 21. Section B.2 presents the impact of data sets for the CPUs, and Section B.2 presents the impact of data sets for the GPUs.

B.1 CPU Performance on Different Data Sets

Figure B.1 helps understand the outcomes for the different data sets against the "Zero" dataset in two ways. First, it considers the performance without terminating the particles (glyphs represented in green) to uncover the impact of caching and memory accesses involved in particle advection. For almost all CPUs and data sets, these experiments performed very close to the ideal case, implying that the impact of caching and memory accesses on CPUs for particle advection is minimal. Second, it considers the performance of terminating the particles normally (glyphs represented in oranges) to uncover the impact of divergence in work for different particles. Again, the experiments performed very close to the ideal case for almost all CPUs data sets, implying that the impact of divergence on CPUs for particle advection is minimal. The anomaly to both these observations was the Fishtank data set, which shows the effects of both poor caching and more divergence. Although Figure 20 demonstrates that while not terminating particles for the Fishtank data set leads to more cache-friendly steps (zero velocity), the particles subjected to this data set also suffer a high variance in the amount of duration contributing to real steps. These variances are the primary reason for this anomaly.

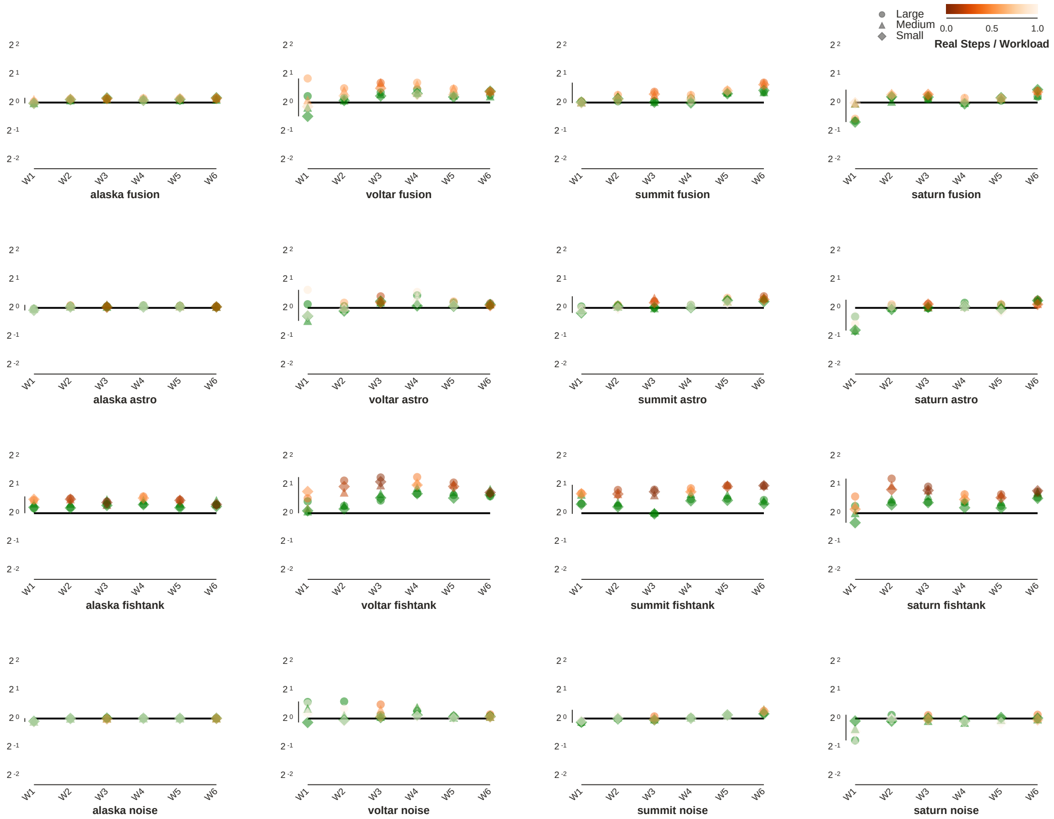


Figure B.1. Comparison of CPU performance for different data sets. The Y-axis represents the ratio of time taken per step against the ideal case (equivalent experiment executed on the “Zero” data set), while the X-axis represents the workload. The different glyphs represent the seeding volume of the experiment. The green dots represent experiments where particles are not terminated at all. In contrast, the orange dots represent experiments where particles are terminated whenever they encounter zero-velocity regions or encounter spatial boundaries. Here a higher Y axis represents the experiment was y times slower than the ideal case.

B.2 GPU Performance on Different Data Sets

Figure B.2 presents a similar analysis as Section , but for GPUs. However, in contrast to CPUs, GPUs performance is impacted significantly due to caching and divergence. Across all the data sets, the two newer GPUs (GPU3 and GPU4) performed poorly in experiments studying caching (green glyphs). In other words, the newer GPUs can offer much better performance when memory accesses for

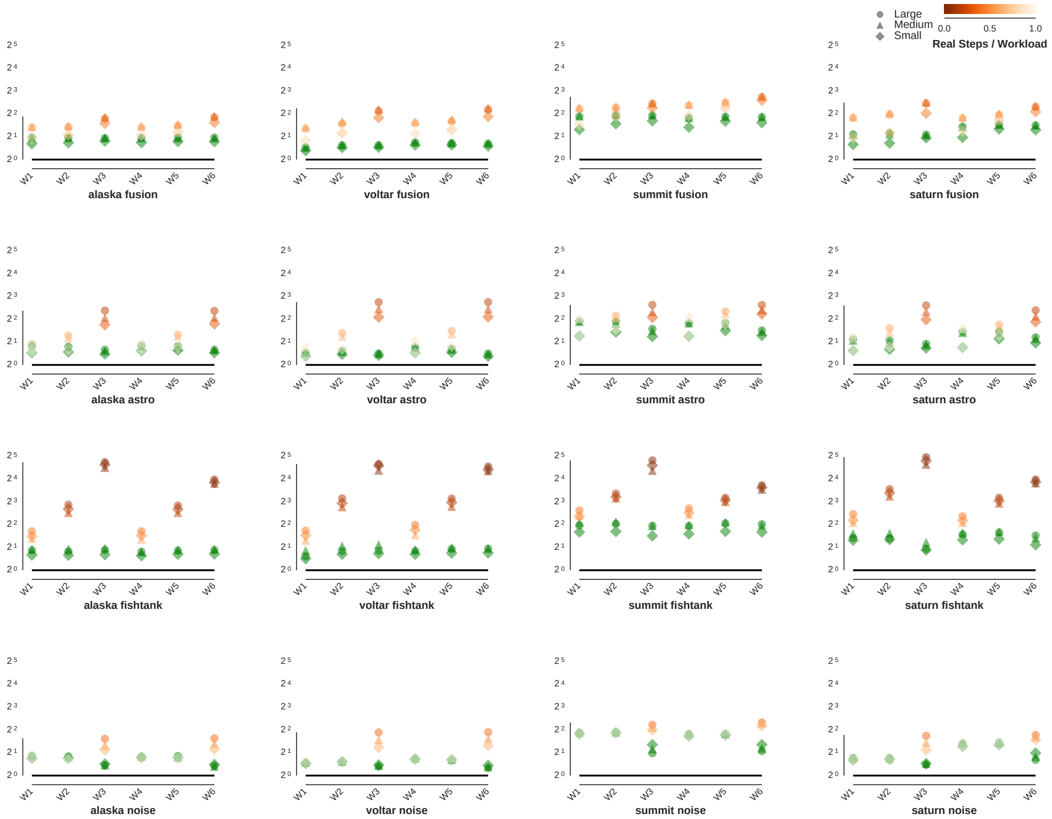


Figure B.2. Comparison of GPU performance for different data sets. The Y-axis represents the ratio of time taken per step against the ideal case (equivalent experiment executed on the "Zero" data set), while the X-axis represents the workload. The different glyphs represent the seeding volume of the experiment. The green dots represent experiments where particles are not terminated at all. In contrast, the orange dots represent experiments where particles are terminated whenever they encounter zero-velocity regions or encounter spatial boundaries. Here a higher Y axis represents the experiment was $\boldsymbol{y}$ times slower than the ideal case.

an application are cache friendly. Particle advection involves random memory accesses, which hurts GPU performance. Further, the performance of a particle advection for a data set is tied to the amount of real work that the algorithm performs proportional to the workload (orange glyphs). For the case of divergence, the takeaway is that a low amount of real work relative to the workload results in poorer performance against the ideal case.

REFERENCES CITED

[1] FieldView CFD. `https://www.fieldviewcfd.com/resources/documentation`, 2022.

[2] Green 500. `https://www.top500.org/lists/green500/2022/06`, 2022.

[3] Nvidia Thrust Library. `https://docs.nvidia.com/cuda/thrust/index.html`, 2022.

[4] Top 500. `https://www.top500.org/lists/top500/2022/06`, 2022.

[5] A. Agranovsky, D. Camp, C. Garth, E. W. Bethel, K. I. Joy, and H. Childs. Improved Post Hoc Flow Analysis via Lagrangian Representations. In *2014 IEEE 4th Symposium on Large Data Analysis and Visualization (LDAV)*, pages 67–75. IEEE, 2014.

[6] J. Ahrens, K. Brislawn, K. Martin, B. Geveci, C. C. Law, and M. Papka. Large-Scale Data Visualization using Parallel Data Streaming. *IEEE Computer Graphics and Applications*, 21(4):34–41, 2001.

[7] O. O. Akande and P. J. Rhodes. Iteration Aware Prefetching for Unstructured Grids. In *2013 IEEE International Conference on Big Data*, pages 219–227. IEEE, 2013.

[8] N. Andrysco and X. Tricoche. Matrix Trees. *Comput. Graph. Forum*, 29(3):963–972, 2010.

[9] U. Ayachit. *The ParaView Guide – ParaView 4.3.* Kitware Inc., 2015.

[10] D. A. Beckingsale, J. Burmark, R. Hornung, H. Jones, W. Killian, A. J. Kunen, O. Pearce, P. Robinson, B. S. Ryujin, and T. R. Scogland. RAJA: Portable performance for large-scale scientific applications. In *2019 Ieee/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, pages 71–81. IEEE, 2019.

[11] N. Bell and J. Hoberock. Thrust – A Productivity-Oriented Library for CUDA. In *GPU Computing Gems – Jade Edition*, pages 359–371. Morgan Kaufmann, 2011.

[12] C. Benthin, I. Wald, S. Woop, M. Ernst, and W. R. Mark. Combining Single and Packet-Ray Tracing for Arbitrary Ray Distributions on the Intel MIC Architecture. *IEEE Transactions on Visualization and Computer Graphics*, 18(9):1438–1448, 2012.

[13] K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, K. Hill, J. Hiller, 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.

[14] R. Binyahib. Scientific Visualization on Supercomputers: A Survey. Available at `http://www.cs.uoregon.edu/Reports/AREA-201903-Binyahib.pdf` (2019/12/12), 2019. Area Exam.

[15] R. Binyahib. Evaluating Parallel Particle Advection Algorithms Over Various Workloads. 2020. Dissertation.

[16] R. Binyahib, D. Pugmire, and H. Childs. In Situ Particle Advection via Parallelizing Over Particles. In *Proceedings of the Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization*, pages 29–33, 2019.

[17] R. Binyahib, D. Pugmire, and H. Childs. HyLiPoD: Parallel Particle Advection Via a Hybrid of Lifeline Scheduling and Parallelization-Over-Data. pages 1–5, 2021.

[18] R. Binyahib, D. Pugmire, B. Norris, and H. Childs. A Lifeline-Based Approach for Work Requesting and Parallel Particle Advection. In *IEEE Symposium on Large Data Analysis and Visualization (LDAV)*, Vancouver, Canada, Oct. 2019.

[19] R. Binyahib, D. Pugmire, A. Yenpure, and H. Childs. Parallel Particle Advection Bake-Off for Scientific Visualization Workloads. In *2020 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 381–391, 2020.

[20] R. Bleile, L. Sugiyama, C. Garth, and H. Childs. Accelerating Advection via Approximate Block Exterior Flow Maps. *Electronic Imaging*, 2017(1):140–148, 2017.

[21] G. E. Blelloch. *Vector Models for Data-Parallel Computing.* MIT Press, 1990.

[22] R. D. Blumofe and C. E. Leiserson. Scheduling Multithreaded Computations by Work Stealing. *Journal of the ACM (JACM)*, 46(5):720–748, 1999.

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

[24] K. Bürger, F. Ferstl, H. Theisel, and R. Westermann. Interactive Streak Surface Visualization on the GPU. *IEEE Transactions on Visualization and Computer Graphics*, (6):1259–1266, 2009.

[25] K. Bürger, F. Ferstl, H. Theisel, and R. Westermann. Interactive Streak Surface Visualization on the GPU. *IEEE Transactions on Visualization and Computer Graphics*, (6):1259–1266, 2009.

[26] K. Burger, P. Kondratieva, J. Kruger, and R. Westermann. Importance-Driven Particle Techniques for Flow Visualization. In *2008 IEEE Pacific Visualization Symposium*, pages 71–78. IEEE, 2008.

[27] K. Bürger, J. Schneider, P. Kondratieva, J. H. Krüger, and R. Westermann. Interactive Visual Exploration of Unsteady 3D Flows. In *EuroVis*, pages 251–258, 2007.

[28] M. Bußler, T. Rick, A. Kelle-Emden, B. Hentschel, and T. W. Kuhlen. Interactive Particle Tracing in Time-Varying Tetrahedral Grids. In *EGPGV*, pages 71–80, 2011.

[29] B. Cabral and L. Leedom. Imaging Vector Fields Using Line Integral Convolution. In *Proceedings of the ACM Conference on Computer Graphics and Interactive Techniques (SIGGRAPH)*, pages 263–270, 1993.

[30] D. Camp, H. Childs, A. Chourasia, C. Garth, and K. I. Joy. Evaluating the Benefits of an Extended Memory Hierarchy for Parallel Streamline Algorithms. In *2011 IEEE Symposium on Large Data Analysis and Visualization*, pages 57–64. IEEE, 2011.

[31] D. Camp, H. Childs, C. Garth, D. Pugmire, and K. I. Joy. Parallel Stream Surface Computation for Large Data Dets. In *IEEE Symposium on Large Data Analysis and Visualization (LDAV)*, pages 39–47. IEEE, 2012.

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

[33] D. Camp, H. Krishnan, D. Pugmire, C. Garth, I. Johnson, E. W. Bethel, K. I. Joy, and H. Childs. GPU Acceleration of Particle Advection Workloads in a Parallel, Distributed Memory Setting. In *Eurographics Symposium on Parallel Graphics and Visualization*. The Eurographics Association, 2013.

[34] C.-M. Chen, B. Nouanesengsy, T.-Y. Lee, and H.-W. Shen. Flow-Guided File Layout for Out-of-Core Pathline Computation. In *IEEE Symposium on Large Data Analysis and Visualization (LDAV)*, pages 109–112. IEEE, 2012.

[35] C.-M. Chen and H.-W. Shen. Graph-Based Seed Scheduling for Out-of-Core FTLE and Pathline Computation. In *2013 IEEE symposium on large-scale data analysis and visualization (LDAV)*, pages 15–23. IEEE, 2013.

[36] C.-M. Chen, L. Xu, T.-Y. Lee, and H.-W. Shen. A Flow-Guided File Layout for Out-of-Core Streamline Computation. In *2011 IEEE Symposium on Large Data Analysis and Visualization*, pages 115–116. IEEE, 2011.

[37] L. Chen and I. Fujishiro. Optimizing Parallel Performance of Streamline Visualization for Large Distributed Flow Datasets. In *2008 IEEE Pacific Visualization Symposium*, pages 87–94. IEEE, 2008.

[38] M. Chen, S. C. Shadden, and J. C. Hart. Fast Coherent Particle Advection through Time-Varying Unstructured Flow Datasets. *IEEE Transactions on Visualization and Computer Graphics*, 22(8):1960–1973, Aug. 2016.

[39] H. Childs, S. Biersdorff, D. Poliakoff, D. Camp, and A. D. Malony. Particle Advection Performance Over Varied Aarchitectures and Workloads. In *2014 21st International Conference on High Performance Computing (HiPC)*, pages 1–10. IEEE, 2014.

[40] H. Childs, E. Brugger, K. Bonnell, J. Meredith, M. Miller, B. Whitlock, and N. L. Max. A Contract Based System for Large Data Visualizations. In *Proceedings ofIEEE Visualization*, pages 191–198. IEEE, 2005.

[41] H. Childs, E. Brugger, B. Whitlock, J. Meredith, S. Ahern, D. Pugmire, K. Biagas, M. C. Miller, C. Harrison, G. H. Weber, H. Krishnan, T. Fogal, A. Sanderson, C. Garth, E. W. Bethel, D. Camp, O. Rubel, M. Durant, J. M. Favre, and P. Navratil. VisIt: An End-User Tool For Visualizing and Analyzing Very Large Data. October 2012.

[42] H. Childs, D. Pugmire, S. Ahern, B. Whitlock, M. Howison, G. H. Weber, E. W. Bethel, et al. Extreme Scaling of Production Visualization Software on Diverse Architectures. *IEEE Computer Graphics and Applications*, (3):22–31, 2010.

[43] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[44] J. Dinan, D. B. Larkins, P. Sadayappan, S. Krishnamoorthy, and J. Nieplocha. Scalable Work Stealing. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–11. IEEE, 2009.

[45] H. C. Edwards and C. R. Trott. Kokkos: Enabling Performance Portability Across Manycore Architectures. In *2013 Extreme Scaling Workshop (XSW 2013)*, pages 18–24. IEEE, 2013.

[46] E. Endeve, C. Y. Cardall, R. D. Budiardja, and A. Mezzacappa. Generation of Magnetic Fields by the Stationary Accretion Shock Instability. *The Astrophysical Journal*, 713(2):1219–1243, Apr 2010.

[47] M. H. Everts, H. Bekker, J. B. Roerdink, and T. Isenberg. Depth-Dependent Halos: Illustrative Rendering of Dense Line Data. *IEEE Transactions on Visualization and Computer Graphics*, 15(6):1299–1306, 2009.

[48] F. Ferstl, K. Burger, H. Theisel, and R. Westermann. Interactive Separating Streak Surfaces. *IEEE Transactions on Visualization and Computer Graphics*, 16(6):1569–1577, 2010.

[49] P. Fischer, J. Lottes, D. Pointer, and A. Siegel. Petascale Algorithms for Reactor Hydrodynamics. *Journal of Physics: Conference Series*, 125:1–5, 2008.

[50] P. Fischer, J. Lottes, D. Pointer, and A. Siegel. Petascale algorithms for reactor hydrodynamics. In *Journal of Physics: Conference Series*, volume 125, page 012076. IOP Publishing, 2008.

[51] M. Fiser. Real Time Visualization of 3D Vector Field with CUDA, 2013.

[52] M. Fullmer and S. Romig. The {OSU} Flow-tools Package and {CISCO}{NetFlow} Logs. In *14th Systems Administration Conference (LISA 2000)*, 2000.

[53] C. Garth, F. Gerhardt, X. Tricoche, and H. Hagen. Efficient Computation and Visualization of Coherent Structures in Fluid Flow Applications. *IEEE Transactions on Visualization and Computer Graphics*, 13(6):1464–1471, 2007.

[54] C. Garth and K. I. Joy. Fast, Memory-Efficient Cell Location in Unstructured Grids for Visualization. *IEEE Transactions on Visualization and Computer Graphics*, 16(6):1541–1550, 2010.

[55] C. Garth, H. Krishnan, X. Tricoche, T. Bobach, and K. I. Joy. Generation of Accurate Integral Surfaces in Time-Dependent Vector Fields. *IEEE Transactions on Visualization and Computer Graphics*, 14(6):1404–1411, 2008.

[56] C. Garth, X. Tricoche, T. Salzbrunn, T. Bobach, and G. Scheuermann. Surface Techniques for Vortex Visualization. In *VisSym*, volume 4, pages 155–164, 2004.

[57] B. Geveci, W. Schroeder, A. Brown, and G. Wilson. VTK. *The Architecture of Open Source Applications*, 1:387–402, 2012.

[58] H. Guo, W. He, S. Seo, H.-W. Shen, E. M. Constantinescu, C. Liu, and T. Peterka. Extreme-Scale Stochastic Particle Tracing for Uncertain Unsteady Flow Visualization and Analysis. *IEEE transactions on visualization and computer graphics*, 25(9):2710–2724, 2018.

[59] H. Guo, F. Hong, Q. Shu, J. Zhang, J. Huang, and X. Yuan. Scalable Lagrangian-Based Attribute Space Projection for Multivariate Unsteady Flow Data. In *2014 IEEE Pacific Visualization Symposium*, pages 33–40. IEEE, 2014.

[60] H. Guo, X. Yuan, J. Huang, and X. Zhu. Coupled Ensemble Flow Line Advection and Analysis. *IEEE Transactions on Visualization and Computer Graphics*, 19(12):2733–2742, 2013.

[61] H. Guo, J. Zhang, R. Liu, L. Liu, X. Yuan, J. Huang, X. Meng, and J. Pan. Advection-Based Sparse Data Management for Visualizing Unsteady Flow. *IEEE transactions on visualization and computer graphics*, 20(12):2555–2564, 2014.

[62] R. Hager, J. Lang, C.-S. Chang, S. Ku, Y. Chen, S. E. Parker, and M. F. Adams. Verification of Long Wavelength Electromagnetic Modes with a Gyrokinetic-Fluid Hybrid Model in the XGC Code. *Physics of plasmas*, 24(5):054508, 2017.

[63] R. Hager, E. Yoon, S. Ku, E. F. D'Azevedo, P. H. Worley, and C.-S. Chang. A Fully Non-Linear Multi-Species Fokker–Planck–Landau Collision Operator for Simulation of Fusion Plasma. *Journal of Computational Physics*, 315:644–660, 2016.

[64] E. Hairer, S. Nørsett, and G. Wanner. *Solving Ordinary Differential Equations I Nonstiff problems*. Springer, Berlin, second edition, 2000.

[65] G. Haller. Distinguished Material Surfaces and Coherent Structures in Three-Dimensional Fluid Flows. *Physica D: Nonlinear Phenomena*, 149(4):248–277, 2001.

[66] M. D. Hanwell, K. M. Martin, A. Chaudhary, and L. S. Avila. The Visualization Toolkit (VTK): Rewriting the rendering code for modern graphics cards. *SoftwareX*, 1:9–12, 2015.

[67] A. Henderson. ParaView Guide: A Parallel Visualization Application. *Kitware, Inc., Clifton Park, NY*, 2007.

[68] B. Hentschel, J. H. Göbbert, M. Klemm, P. Springer, A. Schnorr, and T. W. Kuhlen. Packet-Oriented Streamline Tracing on Modern SIMD Architectures. In *Proceedings of the EG Symposium on Parallel Graphics and Visualization*, pages 43–52, 2015.

[69] D. Hilbert. Über die stetige abbildung einer linie aufein flächenstück. *Mathematische Annalen*, 38:459–460, 1891.

[70] M. Hlawatsch, F. Sadlo, and D. Weiskopf. Hierarchical Line Integration. *IEEE transactions on visualization and computer graphics*, 17(8):1148–1163, 2010.

[71] R. D. Hornung and J. A. Keasler. The RAJA Portability Layer: Overview and Status. 9 2014.

[72] J. P. Hultquist. Constructing Stream Surfaces in Steady 3D Vector Fields. In *Proceedings ofIEEE Visualization*, pages 171–178, 1992.

[73] M. Jiang, B. V. Essen, C. Harrison, and M. B. Gokhale. 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*, pages 11–18. IEEE Computer Society, 2014.

[74] J. Kasten, C. Petz, I. Hotz, H.-C. Hege, B. R. Noack, and G. Tadmor. Lagrangian Feature Extraction of the Cylinder Wake. *Physics of fluids*, 22(9):091108, 2010.

[75] W. Kendall, J. Huang, T. Peterka, R. Latham, and R. Ross. Toward a General I/O Layer for Parallel-Visualization Applications. *IEEE Computer Graphics and Applications*, 31(6):6–10, 2011.

[76] W. Kendall, J. Wang, M. Allen, T. Peterka, J. Huang, and D. Erickson. Simplified Parallel Domain Traversal. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 10:1–10:11, New York, NY, USA, 2011. ACM.

[77] D. N. Kenwright and D. A. Lane. Interactive Time-Dependent Particle Tracing Using Tetrahedral Decomposition. *IEEE Transactions on Visualization and Computer Graphics*, 2(2):120–129, 1996.

[78] J. Krüger, P. Kipfer, P. Kondratieva, and R. Westermann. A Particle System for Interactive Visualization of 3D Flows. *IEEE Transactions on Visualization and Computer Graphics*, 11(6):744–756, 2005.

[79] S. Ku, R. Hager, C.-S. Chang, J. Kwon, and S. E. Parker. A New Hybrid-Lagrangian Numerical Scheme for Gyrokinetic Simulation of Tokamak Edge Plasma. *Journal of Computational Physics*, 315:467–475, 2016.

[80] M. Langbein, G. Scheuermann, and X. Tricoche. An Efficient Point Location Method for Visualization in Large Unstructured Grids. In *Proceedings of Vision, Modeling, Visualization*, 2003.

[81] R. S. Laramee, G. Erlebacher, C. Garth, T. Schafhitzel, H. Theisel, X. Tricoche, T. Weinkauf, and D. Weiskopf. Applications of Texture-Based Flow Visualization. *Engineering Applications of Computational Fluid Mechanics*, 2(3):264–274, 2008.

[82] R. S. Laramee, H. Hauser, H. Doleisch, B. Vrolijk, F. H. Post, and D. Weiskopf. The State of the Art in Flow Visualization: Dense and Texture-Based Techniques. In *Computer Graphics Forum*, volume 23, pages 203–221. Wiley Online Library, 2004.

[83] R. S. Laramee, H. Hauser, L. Zhao, and F. H. Post. Topology-Based Flow Visualization, the State of the Art. In *Topology-based methods in visualization*, pages 1–19. Springer, 2007.

[84] M. Larsen, S. Labasan, P. Navrátil, J. Meredith, and H. Childs. Volume Rendering Via Data-Parallel Primitives. In *Proceedings of the EG Symposium on Parallel Graphics and Visualization*, pages 53–62, 2015.

[85] M. Larsen, J. Meredith, P. Navrátil, and H. Childs. Ray-Tracing Within a Data Parallel Framework. In *Proceedings of the IEEE Pacific Visualization Symposium*, pages 279–286, Hangzhou, China, 2015.

[86] C. Lexi. Application-Specific: Polar, Far-Field, and Particle Tracing Plots, 2015.

[87] R. Liu, H. Guo, J. Zhang, and X. Yuan. Comparative Visualization of Vector Field Ensembles Based on Longest Common Subsequence. In *2016 IEEE Pacific Visualization Symposium (PacificVis)*, pages 96–103. IEEE, 2016.

[88] L. Lo, C. Sewell, and J. Ahrens. PISTON: A Portable Cross-Platform Framework for Data-Parallel Visualization Operators. In *Proceedings of the EG Symposium on Parallel Graphics and Visualization*, 2012.

[89] R. Löhner and J. Ambrosiano. A Vectorized Particle Tracer for Unstructured Grids. *Journal of Computational Physics*, 91(1):22–31, 1990.

[90] K. Lu, H.-W. Shen, and T. Peterka. 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*, pages 1008–1019. IEEE, 2014.

[91] T. McLoughlin, R. S. Laramee, R. Peikert, F. H. Post, and M. Chen. Over Two Decades of Integration-Based, Geometric Flow Visualization. volume 29, pages 1807–1829, 2010.

[92] J. S. Meredith, S. Ahern, D. Pugmire, and R. Sisneros. EAVL: The Extreme-scale Analysis and Visualization Library. In *Proceedings of the EG Symposium on Parallel Graphics and Visualization*, 2012.

[93] K. Moreland, U. Ayachit, B. Geveci, and K.-L. Ma. Dax Toolkit: A proposed framework for data analysis and visualization at Extreme Scale. In *Proceedings of the IEEE Symposium on Large Data Analysis and Visualization*, pages 97–104, 2011.

[94] K. Moreland, R. Maynard, D. Pugmire, A. Yenpure, A. Vacanti, M. Larsen, and H. Childs. Minimizing Development Costs for Efficient Many-Core Visualization Using MCD^3. *Parallel Computing*, 108:102834, Dec. 2021.

[95] K. Moreland, C. Sewell, W. Usher, L. ta Lo, J. Meredith, D. Pugmire, J. Kress, H. Schroots, K.-L. Ma, H. Childs, M. Larsen, C.-M. Chen, R. Maynard, and B. Geveci. VTK-m: Accelerating the Visualization Toolkit for Massively Threaded Architectures. *IEEE Computer Graphics and Applications*, 36:48–58, 2016.

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

[97] C. Müller, D. Camp, B. Hentschel, and C. Garth. Distributed Parallel Particle Advection using Work Requesting. In *2013 IEEE Symposium on Large-Scale Data Analysis and Visualization (LDAV)*, pages 1–6. IEEE, 2013.

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

[99] B. Nouanesengsy, T.-Y. Lee, K. Lu, H.-W. Shen, and T. Peterka. Parallel Particle Advection and FTLE Computation for Time-Varying Flow Fields. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–11. IEEE Computer Society Press, 2012.

[100] B. Nouanesengsy, T.-Y. Lee, and H.-W. Shen. Load-Balanced Parallel Streamline Generation on Large Scale Vector Fields. *IEEE Transactions on Visualization and Computer Graphics*, 17(12):1785–1794, 2011.

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

[102] T. Peterka, R. Ross, B. Nouanesengsy, T.-Y. Lee, H.-W. Shen, W. Kendall, and J. Huang. A Study of Parallel Particle Tracing for Steady-State and Time-Varying flow fields. In *2011 IEEE International Parallel & Distributed Processing Symposium*, pages 580–591. IEEE, 2011.

[103] A. P. Petkov. Transparent Line Integral Convolution: A New Approach for Visualizing Vector Fields in OpenDX. 2005.

[104] A. Pobitzer, R. Peikert, R. Fuchs, B. Schindler, A. Kuhn, H. Theisel, K. Matković, and H. Hauser. The State of the Art in Topology-Based Visualization of Unsteady Flow. In *Computer Graphics Forum*, volume 30, pages 1789–1811. Wiley Online Library, 2011.

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

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

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

[108] D. Pugmire, H. Childs, C. Garth, S. Ahern, and G. H. Weber. Scalable Computation of Streamlines on Very Large Datasets. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, page 16. ACM, 2009.

[109] D. Pugmire, T. Peterka, and C. Garth. Parallel Integral Curves. In E. W. Bethel, H. Childs, and C. D. Hansen, editors, *High Performance Visualization: Enabling Extreme-Scale Scientific Insight*, chapter 6, pages 91–114. Chapman & Hall, 2012.

[110] D. Pugmire, A. Yenpure, M. Kim, J. Kress, R. Maynard, H. Childs, and B. Hentschel. Performance-Portable Particle Advection with VTK-m. In H. Childs and F. Cucchietti, editors, *Eurographics Symposium on Parallel Graphics and Visualization*. The Eurographics Association, 2018.

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

[112] P. J. Rhodes, X. Tang, R. D. Bergeron, and T. M. Sparr. Iteration Aware Prefetching for Large Multidimensional Scientific Datasets. In *Proc. of the 17th international conference on Scientific and statistical database management (SSDBM)*, pages 45–54, 2005.

[113] F. Sadlo and R. Peikert. Efficient Visualization of Lagrangian Coherent Structures by Filtered AMR Ridge Extraction. *IEEE Transactions on Visualization and Computer Graphics*, 13(6):1456–1463, 2007.

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

[115] S. Sane, R. Bujack, C. Garth, and H. Childs. A Survey of Seed Placement and Streamline Selection Techniques. In *Computer Graphics Forum*, volume 39, pages 785–809. Wiley Online Library, 2020.

[116] S. Sane, H. Childs, and R. Bujack. An Interpolation Scheme for VDVP Lagrangian Basis Flows. In *EGPGV@ EuroVis*, pages 109–119, 2019.

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

[118] M. Schirski, C. Bischof, and T. Kuhlen. Interactive Particle Tracing on Tetrahedral Grids Using the GPU. In *Proceedings of Vision, Modeling, and Visualization (VMV) 2006*, pages 153–160, 2006.

[119] M. Schirski, T. Kuhlen, M. Hopp, P. Adomeit, S. Pischinger, and C. Bischof. Virtual Tubelets - Efficiently Visualizing Large Amounts of Particle Trajectories. *Computers & Graphics*, 29(1):17–27, 2005.

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

[121] S. Shiina and K. Taura. Almost Deterministic Work Stealing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '19, pages 47:1–47:16, New York, NY, USA, 2019. ACM.

[122] R. Sisneros and D. Pugmire. Tuned to Terrible: A Study of Parallel Particle Advection State of the Practice. In *Proceedings of the Parallel and Distributed Processing Symposium (IPDPS)*, 2016.

[123] C. Sovinec, A. Glasser, T. Gianakon, D. Barnes, R. Nebel, S. Kruger, S. Plimpton, A. Tarditi, M. Chu, and the NIMROD Team. Nonlinear Magnetohydrodynamics with High-order Finite Elements. *Journal of Computational Physics*, 195(1):355–386, 2004.

[124] D. Sujudi and R. Haimes. Integration of Particles and Streamlines in a Spatially-Decomposed Computation. *Proceedings of Parallel Computational Fluid Dynamics, Los Alamitos, CA*, 1996.

[125] S.-K. Ueng, C. Sikorski, and K.-L. Ma. Efficient Streamline, Streamribbon, and Streamtube Constructions on Unstructured Grids. *IEEE Transactions on Visualization and Computer Graphics*, 2(2):100–110, 1996.

[126] C. Upson, T. A. Faulhaber, D. Kamins, D. Laidlaw, D. Schlegel, J. Vroom, R. Gurwitz, and A. Van Dam. The Application Visualization System: A Computational Environment for Scientific Visualization. *IEEE Computer Graphics and Applications*, 9(4):30–42, 1989.

[127] J. J. van Wijk. Image Based Flow Visualization. *ACM Transactions on Graphics*, 21(3):745–754, 2002.

[128] J.-L. Vay, A. Almgren, J. Bell, L. Ge, D. Grote, M. Hogan, O. Kononenko, R. Lehe, A. Myers, C. Ng, et al. Warp-X: A New Wxascale Computing Platform for Beam–Plasma Simulations. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 909:476–479, 2018.

[129] J.-L. Vay, A. Huebl, A. Almgren, L. Amorim, J. Bell, L. Fedeli, L. Ge, K. Gott, D. Grote, M. Hogan, et al. Modeling of a Chain of Three Plasma Accelerator Stages with the WarpX Electromagnetic PIC Code on GPUs. *Physics of Plasmas*, 28(2):023105, 2021.

[130] L. Wang. On Properties of Fluid Turbulence along Streamlines. *Journal of Fluid Mechanics*, 648:183–203, 2010.

[131] L. Wang and N. Peters. The Length-Scale Distribution Function of the Distance between Extremal Points in Passive Scalar Turbulence. *Journal of Fluid Mechanics*, 554:457–475, 2006.

[132] J. Wilhelms and A. van Gelder. Octrees for Faster Isosurface Generation. *ACM Transactions of Graphics*, 11(3):201–227, 1992.

[133] H. Yu, C. Wang, and K.-L. Ma. Parallel Hierarchical Visualization of Large Time-Varying 3D Vector Fields. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, page 24. ACM, 2007.

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

[135] J. Zhang, H. Guo, F. Hong, X. Yuan, and T. Peterka. Dynamic Load Balancing Based on Constrained K-D Tree Decomposition for Parallel Particle Tracing. *IEEE transactions on visualization and computer graphics*, 24(1):954–963, 2017.

[136] J. Zhang, H. Guo, and X. Yuan. Efficient Unsteady Flow Visualization with High-Order Access Dependencies. In *2016 IEEE Pacific Visualization Symposium (PacificVis)*, pages 80–87. IEEE, 2016.

[137] J. Zhang and X. Yuan. A Survey of Parallel Particle Tracing Algorithms in Flow Visualization. *Journal of Visualization*, 21(3):351–368, 2018.

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