EVALUATING PARALLEL PARTICLE ADVECTION ALGORITHMS OVER VARIOUS WORKLOADS

by

ROBA BINYAHIB

A DISSERTATION

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

March 2020

DISSERTATION APPROVAL PAGE

Student: Roba Binyahib

Title: Evaluating Parallel Particle Advection Algorithms Over Various Workloads

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

Hank Childs	Chair Advisor
Allen Malony	Core Member
Boyana Norris	Core Member
Amanda Thomas	Institutional Representative

and

Kate Mondloch	Vice Provost and Dean of the Graduate School

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

Degree awarded March 2020

© 2020 Roba Binyahib
All rights reserved.

DISSERTATION ABSTRACT

Roba Binyahib

Doctor of Philosophy

Department of Computer and Information Science

March 2020

Title: Evaluating Parallel Particle Advection Algorithms Over Various Workloads

We consider the problem of efficient particle advection in a distributed-memory parallel setting, focusing on four popular parallelization algorithms. The performance of each of these algorithms varies based on the desired workload. Our research focuses on two important questions: (1) which parallelization techniques perform best for a given workload?, and (2) what are the unsolved problems in parallel particle advection? To answer these questions, we ran a "bake off" study between the algorithms with 216 tests, going to a concurrency up to 8192 cores and considering data sets as large as 34 billion cells with 300 million particles. We also performed a variety of optimizations to the algorithms, including fundamental enhancements to the "work requesting algorithm" and we introduce a new hybrid algorithm that we call "HyLiPoD." Our findings inform tradeoffs between the algorithms and when domain scientists should switch between them to obtain better performance. Finally, we consider the future of parallel particle advection, i.e., how these algorithms will be run with in situ processing.

This dissertation includes previously published co-authored material.

CURRICULUM VITAE

NAME OF AUTHOR: Roba Binyahib

GRADUATE AND UNDERGRADUATE SCHOOLS ATTENDED:

University of Oregon, Eugene, OR, USA
King Abdullah University of Science and Technology, Thuwal, Saudi Arabia
King Abdulaziz University, Jeddah, Saudi Arabia

DEGREES AWARDED:

Doctor of Philosophy, Computer and Information Science, 2020, University of Oregon
Master of Science, Computer Science, 2013, King Abdullah University of Science and Technology
Bachelor of Science, Computer Science, 2010, King Abdulaziz University

AREAS OF SPECIAL INTEREST:

Flow Visualization
High Performance Computing
Scientific Visualization

PROFESSIONAL EXPERIENCE:

Graduate Teaching Fellow, University of Oregon, 2020–Present
Graduate Research Fellow, University of Oregon, March 2019 – Dec 2019
Visualization Scientist, National Renewable Energy Laboratory, Summer 2018
Graduate Researcher, Oak Ridge National Laboratory, Summer 2017
Research Aide, Argonne National Laboratory, Summer 2016
Lab Scientist, Saudi ARAMCO, 2013–2014
Lab Technician, King Abdulaziz University, 2010–2011

GRANTS, AWARDS AND HONORS:

Best Paper Honorable Mention at LDAV, 2019
Area Exam Passed With Distinction, University of Oregon, 2019
J. Donald Hubbard Scholarship, University of Oregon, 2018
King Abdullah Scholarship, 2014–2019
KAUST Discovery Scholarship, 2011–2013

PUBLICATIONS:

Roba Binyahib, David Pugmire, Abhishek Yenpure, and Hank Childs. “Parallel Particle Advection Bake-off.” (In preparation.)

Roba Binyahib, David Pugmire, and Hank 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 (ISAV), 2019.

Roba Binyahib, David Pugmire, Boyana Norris, and Hank Childs. “A Lifeline-Based Approach for Work Requesting and Parallel Particle Advection,” In IEEE Symposium on Large Data Analysis and Visualization (LDAV), 2019.

Roba Binyahib, Tom Peterka, Matthew Larsen, Kwan-Liu Ma, Hank Childs. “A Scalable Hybrid Scheme for Ray-casting of Unstructured Volume Data,” In IEEE Transactions on Visualization and Computer Graphics (TVCG), 2018.

Brenton Lessley, **Roba Binyahib**, Robert Maynard, and Hank Childs. “External Facelist Calculation with Data-Parallel Primitives,” In Proceedings of EuroGraphics Symposium on Parallel Graphics and Visualization (EGPGV), 2016.

F Diaz Ledezma, Ayman Amer, Fadl Abdellatif, Ali Outa, Hassane Trigui, Sahejad Patel, and **Roba Binyahib**. “A Market Survey of Offshore Underwater Robotic Inspection Technologies for the Oil and Gas Industry,” In SPE Saudi Arabia Section Annual Technical Symposium and Exhibition, 2015.

ACKNOWLEDGMENTS

First, I would like to thank my advisor, Dr. Hank Childs, for his continuous support and guidance. Dr. Childs was a great advisor. He taught me how to be a better researcher, how to review papers, how to write funding proposals, and how to be a better programmer. He always encouraged me to learn new skills and helped me to pursue opportunities for growth. During our first meeting, he asked me what my professional goals are. Since then, he made sure to teach me all the necessary skills to reach my goals. Thank you for all the time and effort. I'm very grateful to have such a great advisor.

I would like to thank my committee members, Dr. Allen Malony, Dr. Boyana Norris, and Dr. Amanda Thomas, for all the help and feedback.

I spent my summers working for different national labs under the supervision of wonderful mentors. I would like to thank Dr. Tom Peterka, Dr. Dave Pugmire, and Dr. Kenny Gruchalla for making my internship experience productive and fun.

I would not be here today if it was not for Dr. Madhu Srinivasan. Thank you for teaching me about scientific visualization and research. Thank you for your patience, especially at the beginning of my research journey. I'm grateful for everything you taught me, for the support, and for introducing me to Dr. Hank Childs.

My time here was great thanks to my CDUX colleagues. Thank you for the support, feedback, and friendship.

During my years in Oregon, I met friends who became family. I enjoyed all the nights we spent studying in the library, all the adventures we have gone through together, and the holidays we celebrated as a family. Thank you for all the joyful and memorable moments.

This research is funded in part by the King Abdullah Scholarship represented by the Saudi Arabian Cultural Mission (SACM). 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. This research was also supported by the Scientific Discovery through Advanced Computing (SciDAC) program of the U.S. Department of Energy. This research used resources of the National Energy Research Scientific Computing Center (NERSC), a U.S. Department of Energy Office of Science User Facility operated under Contract No. DE-AC02-05CH11231.

I dedicate this dissertation to my family. Thank you for your unconditional love and support. Thank you for always being there for me, even when we are on different sides of the world. You taught me the value of hard work, encouraged me to achieve my goals, and always cheered me up. I would not have been here if it was not for you.

TABLE OF CONTENTS

Chapter Page

I Foundations 1

I. INTRODUCTION 3
  1.1. Dissertation Plan 5
  1.2. Dissertation Outline 7
  1.3. Abbreviations 8
  1.4. Co-Authored Material 9

II. BACKGROUND 10
  2.1. Scientific Visualization in a Distributed Memory Setting 11
  2.2. Scalar Field Visualization 16
  2.3. Supporting Infrastructure 33

III. PARALLEL PARTICLE ADVECTION ALGORITHMS 45
  3.1. Foundations 45
  3.2. Studied Parallel Particle Advection Algorithms 48
  3.3. Other Parallel Particle Advection Algorithms 51

II Improving Individual Parallel Particle Advection Algorithms 60

Chapter Page

IV. BEST PRACTICES AND IMPROVEMENTS TO THE PARALLEL ALGORITHMS 62

4.1. Parallel Particle Advection Best Practices 62

4.2. A Lifeline-Based Approach for Work Requesting and Parallel Particle Advection 63

III Understanding Parallel Particle Advection Behavior Over Various Workloads 90

V. PARALLEL PARTICLE ADVECTION BAKE-OFF 92

5.1. Motivation 92

5.2. Experiment Overview 92

5.3. Testing Infrastructure 97

5.4. Results 100

5.5. Summary of Findings 116

VI. HYLIPOD: IMPROVED HYBRID PARALLEL PARTICLE ADVECTION ALGORITHM 118

6.1. Motivation 118

6.2. Algorithm 118

6.3. Experiments Overview 119

6.4. Results 119

IV The Future of Parallel Particle Advection 123

Chapter Page

VII. IN SITU PARALLEL PARTICLE ADVECTION 125

7.1. Motivation 125

7.2. Related Work 127

7.3. Algorithm 127

7.4. Experimental Overview 129

7.5. Results 133

7.6. Conclusion 135

VIII. CONCLUSION AND FUTURE WORK 137

8.1. Synthesis 137

8.2. Recommendations for Future Study 139

REFERENCES CITED 142

LIST OF FIGURES

Figure Page

1. A visualization pipeline using the data flow design. 14

2. Volume rendering via ray-casting. 17

3. Example execution of a hybrid volume rendering algorithm. 25

4. Example of structured and unstructured meshes. 27

5. Example of applying a contouring algorithm on an AMR grid. 32

6. Image compositing using the Direct Send method between four processors. 36

7. Image compositing using the Binary Swap method between four processors. 37

8. Image compositing using the Radix-k method between six processors. . . 38

9. The distribution of work in the two main parallel particle advection algorithms. 47

10. Different flow visualization algorithms that use particle advection 56

11. A Lifeline graph of four nodes. 65

12. Streamline visualization of the four data sets used in the study of the new work requesting algorithm introduced in Chapter IV. . 70

13. Performance of the four algorithms considered in the study of the new work requesting algorithm introduced in Chapter IV. 74

14. Performance of the four algorithms considered in the study of the new work requesting algorithm introduced in Chapter IV using different data sets 78

15. The three main factors used in the study in Chapter V 93

16. The three seeding boxes considered in the study in Chapter V 95

17. The performance scalability of the parallelize over data algorithm for different workloads. 102

Figure Page

18. The performance scalability of the parallelize over data algorithm for different workloads. 103

19. The performance scalability of the parallelize over particles algorithm for different workloads. 105

20. The performance scalability of the parallelize over particles algorithm for different workloads. 106

21. The performance scalability of the work requesting algorithm for different workloads. 108

22. The performance scalability of the work requesting algorithm for different workloads. 109

23. The performance scalability of the master/worker algorithm for different workloads. 111

24. The performance scalability of the master/worker algorithm for different workloads. 112

25. Comparing the performance of the four algorithms for different workloads considered in Chapter V. 114

26. The performance scalability of our hybrid parallel particle advection algorithm (HyLiPoD) for different workloads. 120

27. Comparing the performance of the three algorithms for different workloads considered in Chapter VI. 122

28. Streamline visualization for the two seed distributions considered in the study in Chapter VII 132

29. Performance of the in situ implementation of the two traditional parallel particle advection algorithms for a dense distribution of seeds 133

LIST OF TABLES

Table Page

1. Factors impacting the performance of parallel volume rendering, and the best configuration for each of these factors using different parallelization techniques. 27
2. Comparing the performance of the four algorithms considered in the study of the new work requesting algorithm introduced in Chapter IV 75
3. Comparing the number of advection steps and I/O operations between the four algorithms considered in the study of the new work requesting algorithm introduced in Chapter IV . . 76
4. Comparing the performance of the four algorithms considered in the study of the new work requesting algorithm introduced in Chapter IV when varying the data sets 78
5. Comparing the performance of the four algorithms considered in the study of the new work requesting algorithm introduced in Chapter IV when varying the number of particles 80
6. Comparing the performance of the four algorithms considered in the study of the new work requesting algorithm introduced in Chapter IV when varying the duration of particles 81
7. Comparing the performance of the four algorithms considered in the study of the new work requesting algorithm introduced in Chapter IV when varying the number of blocks 82
8. Comparing the performance of the four algorithms considered in the study of the new work requesting algorithm introduced in Chapter IV when varying the number of cells per block 84
9. Comparing the performance scalability of the four algorithms considered in the study of the new work requesting algorithm introduced in Chapter IV 85

Table Page

10. Comparing the difference in workload balance between the four algorithms considered in the study of the new work requesting algorithm introduced in Chapter IV 85

11. The best case for each of the four algorithms considered in the Bake-off study (Chapter V). 113

12. The best algorithm for the different seeding box sizes considered in the Bake-off study (Chapter V). 116

13. Comparing the performance of HyLiPoD to LSM and POD (Chapter VI). 121

14. Comparing the performance and memory consumption of the two algorithms considered in Chapter VII for a dense particle distribution 134

15. Comparing the performance and memory consumption of the two algorithms considered in Chapter VII for uniform seeding 135

Part I

Foundations

This part of the dissertation discusses the motivation of this work, and then provides background of scientific visualization on supercomputers.

CHAPTER I

INTRODUCTION

Simulations enable scientists to study complex phenomena which may have been too difficult or expensive to study experimentally. That said, simulations can only replace experiments if they have sufficient accuracy, and achieving this accuracy often requires fine mesh resolution. Yet, working with such data requires high computational power and large memory; requirements that go far beyond the capabilities of a single machine. Supercomputers allow scientists to achieve finer mesh resolutions by performing calculations at a massive scale. Examples of fields that regularly use large scale simulations are high energy physics, biology, and cosmology. Simulations in these fields, and others, produce data sets potentially containing trillions of cells. These massive amounts of data are the key to future discoveries and scientific breakthroughs. Further, visualization is a powerful tool to achieve that goal, enabling scientists with ways to explore, extract, understand, and analyze important information.

Many of these simulations generate vector data, encoding phenomena such as the formation of wind turbine wakes, biomass pyrolysis, or efficiencies in vehicle platooning. The behavior and patterns occurring in these vector flows can be understood using the subset of visualization techniques dedicated to vector data, called "flow visualization." There are myriad flow visualization algorithms (described in Chapter II), and these algorithms typically rely on the same operation as a building block: particle advection. Advection is the process of displacing a massless particle depending on the vector field. The trajectory of each particle as it is displaced can be described by an ordinary differential equation. In practice, this trajectory is calculated iteratively. A particle with an initial (seed) location,

X_0, is displaced to a nearby location, X_1, then displaced again to another location, X_2, and so on. Each advancement, i.e., from X_i to X_{i+1}, is referred to as a step. The change in position for a given step is typically approximated using numerical methods, such as Runge-Kutta [1]. These numerical methods require multiple vector field evaluations at different spatial locations (and sometimes temporal locations for time-varying flow). A flow visualization technique will then use the computed trajectories of each particle to display its representation of features within the flow field. The representations used by each technique are varied. Some simply plot the position or trajectory of particles, while others create derived quantities based on trajectory properties.

Because the different flow visualization techniques are highly varied, the corresponding particle advection workloads are similarly highly varied. The number of particles can be as few as one to potentially billions. Further, the number of steps can vary as well, from under one hundred steps to hundreds of thousands of steps, or more. As a result, some particle advection workloads have excessive computation times — as many as trillions of steps, with each step requiring velocity field interpolations to solve an ordinary differential equation (Runge-Kutta).

Particle advection solutions get considerably more complex in the context of supercomputers. This setting typically adds two significant complications: (1) data sets contain many cells and are decomposed into blocks, and (2) the number of advection steps to calculate is so large that parallel processing is required. Supercomputer architectures add to the complexity as well, as they are made up of many nodes, with each node containing its own (private) memory. Ultimately, the fundamental challenge of efficient parallel particle advection on supercomputers is to make sure the correct particle and vector field information are together at the same time so a step can occur. Unfortunately, data set sizes often preclude the simplest method for achieving this solution — loading all vector field information on all nodes.

The visualization community has introduced several parallel solutions to address these challenges. However, there is no comprehensive comparison identifying the suitable techniques for specific workloads. In this dissertation, we evaluate and compare the most popular parallel particle advection techniques. We implement and improve the different algorithms and conduct a comprehensive study to answer the following questions:

- **Which parallelization technique performs best for a given workload?**
- **What are the unsolved problems in parallel particle advection? Are there any workloads that are difficult to balance using existing parallelization techniques?**

1.1 Dissertation Plan

This section describes the three studies contributing to the dissertation, which are:

- Study 1: Optimizing Performance of Each Parallelization Algorithm.
- Study 2: Comparing Behavior of the Algorithms Over Various Workloads.
- Study 3: Considering the Future of Parallel Particle Advection.

In this dissertation, we compare four of the most used parallel algorithms (parallelize-over-data, parallelize-over-particle, work-requesting, and master-worker, see Chapter III for more details) across various workloads. Our goals are to help end users select the best parallelization algorithm for their workload, and to inform the visualization community regarding opportunities for improvement.

1.1.1 Study 1: Optimizing Performance of Each Parallelization Algorithm. We studied the four parallelization algorithms and their implementations and looked for possible improvements. For all four algorithms, we added on node parallelism using VTKm [2, 3]. We also added an improvement to the work requesting algorithm by replacing the random scheduling method [4] with the Lifeline scheduling method [5]. Lifeline is currently the high-performance computing community's preferred scheduling method for work requesting [5, 6, 7].

1.1.2 Study 2: Comparing Behavior of the Algorithms Over Various Workloads. Our main study is a bake off that evaluates and compares the four parallelization algorithms we consider. We created a platform that allows the change of different factors, which helps to study different cases. From our study of state of the art (see Chapter III), we determined three different factors that have the highest impact on the performance. These factors are: 1) seed distribution method, 2) number of seeds, and 3) concurrency. We test the cross product of these factors and analyze the results to determine how these factors impact each algorithm. In addition to this study, we also implemented a new fifth algorithm that is a hybrid between two of the existing algorithms.

1.1.3 Study 3: Considering the Future of Parallel Particle Advection. In situ visualization is a promising solution to reduce the cost of I/O by visualizing the simulation as it is running, avoiding intermediate data files. In situ visualization methods usually adopt a tightly coupled approach, where the simulation and visualization are executed synchronously on the same computation resources in a time sharing manner. One significant challenge in the situ setting is limitations in memory usage.

While several solutions have been proposed for parallel particle advection in a post hoc setting, in situ solutions tends to use the parallelize over data technique. That is because this technique aligns with in situ constraints. In this study, we explored whether other parallelization techniques are suitable for tightly-coupled in situ processing as well. In particular, we adapted the parallelize over particles technique to work in an situ setting and compared both methods with different seed placements.

1.2 Dissertation Outline

This dissertation is organized into the following four parts:

- Part I – Foundations
    - Chapter I discusses the motivation behind this work, and describes the dissertation questions and plan.
    - Chapter II surveys the research done on visualization techniques in a distributed memory setting.
    - Chapter III describes the foundations of particle advection, the parallel particle advection algorithms that are considered in this dissertation, as well as surveying the research done on parallel particle advection.
- Part II – Improving Individual Parallel Particle Advection Algorithms
    - Chapter IV discusses the best practices that were adopted and implemented from previous studies, and introduces an improvement for the work requesting algorithm.

- Part III – Understanding Parallel Particle Advection Behavior Over Various Workloads
    - Chapter V presents our bake off study, where we test the different parallel particle advection algorithms over different workloads.
    - Chapter VI introduces a new hybrid algorithm that adapts its behavior depending on the workload characteristics.
- Part IV – The Future of Parallel Particle Advection
    - Chapter VII explores in situ parallel particle advection by studying the two main algorithms over two different workloads.
    - Chapter VIII concludes this dissertation and discusses the lessons learned and suggests recommendations for future research.

1.3 Abbreviations

This is a list of the abbreviations used in this dissertation.

- **POD**: Parallelize-Over-Data algorithm (defined in Section 3.2.1)
- **POP**: Parallelize-Over-Particles algorithm (defined in Section 3.2.2)
- **RSM**: Work requesting algorithm using the Random Scheduling Method with a single victim (defined in Section 4.2.1)
- **RSM-N**: Work requesting algorithm using the Random Scheduling Method with multiple victims (defined in Section 4.2.1)
- **LSM**: Work requesting algorithm using the Lifeline Scheduling Method (defined in Section 4.2.1)

- **MW**: Master/Worker algorithm (defined in Chapter 3.2.4)
- **HyLiPoD**: Our proposed Hybrid Lifeline Parallelize-Over-Data particle advection algorithm, which is a hybrid between LSM and POD (defined in Section 5.1)

1.4 Co-Authored Material

Most of the work in this dissertation is from previously published co-authored research. The following is a description of the chapters with the publications and authors that contributed to it. Additional details on the division of work is provided at the beginning of each chapter

- Chapter I: parts of the text in this chapter comes from the introduction of [8], which was a collaboration between David Pugmire (ORNL), Boyana Norris (UO), Hank Childs (UO), and myself.
- Chapter II: comes from my Ph.D. Area Exam document, which was unpublished.
- Chapter IV: parts of the text in this chapter comes from the introduction of [8], which was a collaboration between David Pugmire (ORNL), Boyana Norris (UO), Hank Childs (UO), and myself.
- Chapter V: comes from a Manuscript in Preparation that was a collaboration between David Pugmire (ORNL), Abhishek Yenpure (UO), Hank Childs (UO), and myself.
- Chapter VII: comes from [9], which was a collaboration between David Pugmire (ORNL), Hank Childs (UO), and myself.

CHAPTER II

BACKGROUND

Parts of the text in this chapter came from my area exam, which received editing suggestions from Hank Childs.

The size of the data sets produced by today's large scale simulations make visualization difficult for several reasons. One complication is that data transfer is expensive, which often prevents transfers to local desktops or visualization clusters. Another complication is in the processing of large data. Some techniques reduce the processing costs by focusing on coarser versions of the data or on subsets of the data. These techniques, including multiresolution techniques and streaming, are used regularly in non-HPC environments. However, in the context of supercomputing, the dominant processing technique is parallelism, i.e. using the same supercomputer for not only simulation but also visualization. This is done by distributing data or workloads across multiple nodes, where each node visualizes its assigned portion. In most cases, the processing is done at the native mesh resolution and storing the entire mesh in memory (although distributed), requiring significant computational power, large memory, and I/O bandwidth. These requirements are often acceptable, however, since performing visualization on a supercomputer allows visualization algorithms to take advantage of the supercomputer resources. That said, visualizing such large data on a supercomputer (i.e., a distributed memory setting) adds new challenges. Even though most visualization algorithms are embarrassingly parallel, others require heavy communications and coordination. In addition, load balance must be maintained to run these algorithms efficiently, even with embarrassingly parallel algorithms.

These challenges have been the subject of various research efforts to improve the scalability and efficiency of visualization algorithms. In this chapter, we cover techniques for visualizing large data sets at scale with an exclusive focus on distributed memory parallelism algorithms and their challenges. We exclude from this chapter the research done on particle advection, since it receives special treatment in Chapter III, as the focal point of this dissertation. The organization of this chapter is as follows. Section 2.1 provides areas of background for scientific visualization in a distributed memory setting. Section 2.2 discusses the research done on visualization techniques for scalar field data. Finally, Section 2.3 discusses supporting infrastructures used by visualization algorithms.

Explicitly, this chapter focuses on performing visualization algorithms on supercomputers, and in particular the methods and optimizations required to visualize large data in a distributed memory setting. Related topics to this chapter include multiresolution processing, streaming, hybrid parallelism, and in situ processing; while these topics are discussed when relevant to visualization on supercomputers, they are otherwise considered out of scope.

2.1 Scientific Visualization in a Distributed Memory Setting

In this section, we cover important areas of background for scientific visualization in a distributed memory setting. We start by discussing the impact of I/O on the visualization pipeline (Section 2.1.1). Next, we discuss the processing techniques for visualization algorithms (Section 2.1.2). Then, we take a look at the framework design used in most of visualization tools (Section 2.1.3). Finally, we discuss the parallelization design of visualization algorithms (Section 2.1.4).

2.1.1 I/O in Scientific Visualization.

Computational power is increasing tremendously, while I/O systems are not improving at nearly the same pace. The main limiting factor for large scale visualization performance is I/O [10, 11]. Several techniques have been proposed to reduce the cost of I/O operations for visualization algorithms such as multiresolution techniques [12, 13], subsetting [14, 15], or parallel processing. In multiresolution techniques, data sets are stored in a hierarchical structure, and visualization is performed starting from the coarser data up to the finer ones. Subsetting is used to read and process only the portion of the data that will contribute to the visualization result. In parallel processing, the visualization method use the computational power of multiple nodes to process the data faster. Despite the presence of the first three techniques, the supercomputing community use parallel processing.

As supercomputers are pushing toward exascale, the gap between computation power and I/O is expected to increase even more. Consequently, I/O constraints are an important factor to take into account when designing visualization systems. Each one of the above mentioned techniques addresses I/O constrains. Multiresolution and subsetting solutions reduce the required I/O. Parallel processing increases the available I/O bandwidth.

2.1.2 Processing Technique.

There are two processing techniques for visualization algorithms: post-hoc and in situ. The traditional paradigm is post-hoc processing, where scientists visualize their data as a post-processing step. In this model, the simulation code saves data to disk and is either read back later on the same computational resources or transferred to another machine for visualization. An alternative solution to reduce the cost of I/O is to use in situ processing [16], where the visualization is performed while the simulation is running. The data is streamed from the simulation code to the visualization. In situ visualization adds new challenges to both simulation codes and visualization systems which must be addressed. These challenges include for instance code modification, data flow management, synchronization between tasks, and difference of data models between the simulation and the visualization tool. Successful examples of in situ systems include Catalyst [17], and Libsim [18], which work along with Paraview and VisIt respectively.

The remainder of this survey will focus on efficient parallelization techniques regardless of their processing model.

2.1.3 Data Flow Framework.

Parallel visualization frameworks have been developed to help users visualize their data. These frameworks include VTK [19], AVS [20], MegaMol [21], VisIt [22], and Paraview [23]. Most of these systems implement a data flow framework. A data flow framework executes a pipeline of modules where a module is an operation on its input data, and a link between two modules is a data stream. A module in the pipeline can be: 1) a source, 2) a filter, or 3) a sink. A source is a module that generates data, usually by reading data from a file. A filter is a module that takes data as an input, applies an operation, and produces an output. A sink is a module that receives data and produces a final result which can be written to file or displayed on a screen. These frameworks implement each visualization algorithm as an independent module. Figure 1 shows an example of a visualization pipeline: the source (read operation) reads data from a file, the filters (compute density, clip data, and compute isosurface) apply operations on the data and generate new data, and the sink (write operation) receives the data to produce an output.

Using a data flow framework has several advantages:

- The framework is abstract and hides the complexity from the users

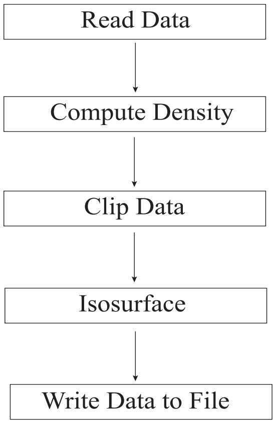


Figure 1. A visualization pipeline using the data flow design.

- The framework is flexible and allows users to add new modules without requiring to modify old modules.
- Modules of the framework can be combined to create advanced analysis.

2.1.4 Parallelization Design.

The main challenge for parallel visualization algorithms is to decompose the work into independent segments, where processors can process their segments in parallel. These segments are usually data blocks. Most of visualization systems use a scatter-gather design. In this design, segments are scattered across different processors. Each processor reads its segment and applies the visualization pipeline using data flow network. Each processor has an identical data flow network and processors differ in the segments they operate on. Then the results of different processors are gathered in the rendering phase.

Visualization algorithms can be classified int two categories: 1) embarrassingly parallel and 2) non-embarrassingly parallel. In an embarrassingly parallel algorithm, each processor can apply visualization on its segment independently. On the other hand, a non-embarrassingly parallel algorithm depends on other processor's computations. The majority of visualization algorithms are embarrassingly parallel.

2.1.5 Load Balance.

A major challenge when running algorithms in parallel is maintaining load balance. Load balance is defined as the allocation of the work of single application to processors so that the execution time of the application is minimized [24]. Maintaining load balance is essential to achieve good performance since the execution time is determined by the time of the slowest processor. There are two categories of load balancing: 1) static, and 2) dynamic. In static load balancing, the workload is distributed among processors during the initialization phase. The workloads then remain on their computational resources during the entire execution of the visualization. The challenge in static load balancing is to guarantee equal workload, which can be difficult for some visualization algorithms. In dynamic load balancing, the workload is distributed during run time by a processor acting as the master. Dynamic load balancing can be used when the workload is unknown before run time.

Most of the solutions in this survey focus on solving load imbalance for different visualization algorithms. Load imbalance can be defined with the following equation:

$$\text{Load imbalance} = \frac{\mathrm{T_s}}{\sum_{0<\mathrm{p}<\mathrm{N}} \mathrm{T_p}/\mathrm{N}}$$

Where T_p is the total non-idle execution time for processor P, and T_s is the total non-idle execution time of the slowest processor.

Load balance is a major focus of this survey as many of the solutions and optimizations were suggested to maintain load balance.

2.2 Scalar Field Visualization

2.2.1 Volume Rendering.

There are two types of rendering: 1) surface rendering, and 2) direct rendering. Surface rendering is generating an image from a geometry that was produced by the visualization pipeline by converting the geometry into pixels through rasterization [25], or ray tracing [26]. Direct volume rendering is generating an image directly from the data using ray-casting [27]. This is done by sampling and mapping samples into color and opacity using a transfer function. In this section, we discuss direct volume rendering.

2.2.1.1 Ray Casting.

Ray casting is commonly used due to its simplicity and the quality of the results. For each pixel in the screen, a ray is cast into the volume and samples are computed along the ray. Next, each sample is mapped into a color and opacity (RGBA values) using the transfer function [28]. These RGBA values are accumulated to compute the final color of the pixel. The accumulation process can be performed either in a front-to-back order or in back-to-front order. Equation 2.1 and 2.2 presents a front-to-back and back-to-front order accumulation, respectively.

$$C = \sum_{i=0}^{n} C_i \prod_{j=0}^{i-1} (1 - A_i) \tag{2.1}$$

Where C is the RGBA value of the pixel, C_i is the color of the current scalar value at sample i, n is the number of samples along the ray, and A_i is the opacity at sample i.

$$C = \sum_{i=n}^{0} C_i \times (1 - A) \tag{2.2}$$

Where C is the RGBA value of the pixel, C_i is the color of the current scalar value at sample i, n is the number of samples along the ray, and A is the accumulated opacity along the ray.

Figure 2 shows an example of the ray-casting process.

Ray-casting is expensive, thus different acceleration techniques have been used to reduce this cost. One of the most used acceleration techniques is early ray termination [29]. Ray casting computes the color of the pixel by accumulating the colors and opacities of the samples along the ray. If the accumulated opacity is high, samples that are far from the camera will not contribute to the final color and will be hidden. The idea of early termination is to stop the compositing along the ray when the accumulated opacity is high, which reduces the total time. However, this optimization is only possible with front-to-back compositing.

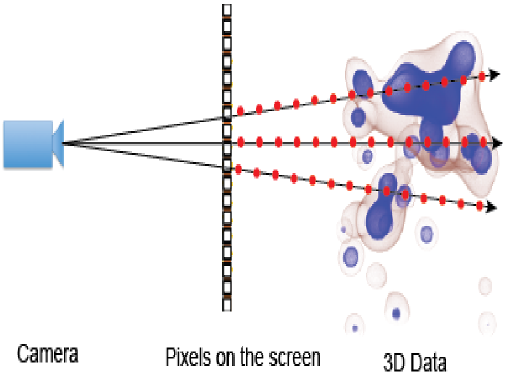


Figure 2. Volume rendering via ray-casting.

2.2.1.2 Parallelization Overview. Volume rendering is computationally expensive, and its cost increases with the size of the data set. Parallelizing such heavy computation is essential to visualize data in a timely manner. However, performing parallel ray-casting introduces new challenges, especially with respect to load balancing (Section 2.1.5). There are two main techniques for parallel volume rendering [30]: 1) image order (sort first), and 2) object order (sort last). In the image order technique, the parallelization happens over pixels. In the object order technique, the parallelization happens over cells (sub-volumes). In this section, we start by discussing the challenges of parallel volume rendering. Next, we survey the different parallel solutions and categorize them under one of three categories: 1) image order (sort first), 2) object order (sort last), and 3) a hybrid between the first two.

The performance of ray-casting depends on two components: 1) the number of cells, and 2) the number of samples. These two components are heavily impacted by four factors, each of which can cause significant load imbalances and influence the choice of parallelization method. These four factors are the following: 1) camera position, 2) camera view is changing, 3) image size, and 4) data Set size.

- **Camera Position:** It impacts the performance in two points: 1) which part of the data is visible, and 2) the number of samples per cells (cell sizes). If the camera is zoomed in, it implies: 1) there are no empty pixels, and 2) cells that are in the camera view have more samples (larger cells). If the camera is zoomed out, it implies: 1) there are empty pixels, and 2) cells have a similar number of samples (equal sizes). Image order performs well when the camera is zoomed in since there are no empty pixels. However, it performs poorly when the camera is zoomed out since there are parts of the image that are empty. On the other hand, object order performs well when the camera is zoomed out because the cells are distributed evenly among processor and most of the cells are in the camera view. However, it can suffer from load imbalance when the camera is inside the volume because only the processors having visible cells (in the camera view) will do the work (larger cells).

- **Moving Camera View :** If the camera view is changing between frames, the visible portion of the data changes between frames. The image order technique is expensive with this configuration because it requires to redistribute data blocks among processors for every new camera view. In some cases, the data is replicated to avoid redistributing the blocks, but this becomes challenging when the size of the data is large and cannot fit into a single memory. On the other hand, object order works well for cases where the camera view frequently changes since each processor works on its cells independently from the camera view.

- **Image Size:** In order to produce the final pixel color, a processor needs to have all the data required for that pixel. In image order, each processor has the data required to produce its part of the image; no exchange is needed between processors. In object order, processors need to exchange samples (i.e., image compositing) to calculate the final color of the pixel. The communication cost of this step is expensive and could become a bottleneck when the size of the image is large. Thus image order works better than object order for large image sizes.

- **Data Set Size:** If the data size is small enough to fit into a single memory, data can be replicated when using image order. As the size increases, using image order becomes difficult and could add additional costs of redistributing data blocks. Object order offers scalability when the data size is large.

2.2.1.3 Image Order. In the image order technique, pixels are distributed among processors in groups of consecutive pixels, also known as tiles. Each processor is responsible for loading and sampling the cells that contribute to its tile. Then, each processor generates a sub-image corresponding to its tile. The sub-images from all the processors are then collected onto one processor to produce the final image.

This technique allows each processor to generate its sub-image independently, avoiding the communication cost of image compositing. Load imbalance can occur if processors have un-equal cell distribution. This can happen when some tiles have more cells than others, which means some processors are performing more work than others, resulting in load imbalance. Different solutions have been proposed to avoid load imbalance by introducing additional steps to guarantee equal cells distribution.

Samanta et al. [31] presented a solution that reduced the probability of un-equal cell distribution by using virtual tiles. These virtual tiles are flexible in their shapes and size depending on the workload. Their solution maintained load balance by assigning similar cell load to each processor.

Erol et al. [32] used a dynamic load balancing method to maintain load balance. Their algorithm divided the workload into tiles and used the previous rendering times to distribute the tiles among processors.

Moloney et al. [33] reduced load imbalance by introducing a bricking step. In this step, the data is divided into bricks, and bricks outside the view frustum are excluded. Next, the view frustum is divided between processors and each processor sampled the bricks within its view. Using the bricking step divides the visible part of the image among processors and eliminates assigning a processor an empty tile.

While the previous solutions maintained load balance which improved the performance, all of these solutions needed a pre-processing step and some included redistribution of the data. Both [31] and [32] required a pre-processing step to determine the load of different tiles, and have the cost of redistributing the data. The third solution, [33], required performing camera transformation to determine visible data, which avoided the cost of redistributing the data.

2.2.1.4 Object Order. Object order is the most common technique for parallel volume rendering. With the object order approach, data is divided into blocks and distributed among processors. Each processor starts sampling the cells of its blocks independently of the other processors. Next, samples from all processors are composited to produce the final image.

Unlike the image order technique, this technique requires processors to communicate with each other to do the final compositing (i.e., image compositing), which could become a bottleneck [34]. Load imbalance can occur if processors have un-equal samples distribution. This can happen when dealing with unstructured data. Unstructured data have different cell sizes creating different workloads: one processor could have large cells thus more workload. Different solutions have been proposed to avoid load imbalance by introducing additional steps to guarantee equal samples distribution.

Marchesin et al. [35] presented a solution to guarantee load balance by performing an estimation step. In their solution, they divided data into blocks and discard any blocks that were outside the camera view or blocks that were invisible. Next, the remaining blocks were distributed among processors, and each processor sampled its blocks. Finally, binary swap [36] was used as an image compositing method.

Ma et al. [37] presented a solution that used round robin cells assignment to perform interleaved cell partitioning. This assignment reduces the probability of load imbalance since usually, cells that are spatially close have similar sizes.

Assigning these cells to different processors helps to avoid heavy workload for some processors. In addition, this assignment achieved load balance when the camera is zoomed into a region of the data. Samples from different processors are stored in a linked list. To allow for early compositing of the samples, processors sample the cells in the same region at the same time.

Steiner et al. [38] achieved load balance by using a work package pulling mechanism [39]. In their solution, work was divided into equal packages and inserted into a queue. Clients asked the server for work whenever they are done with their assigned workload.

Muller et al. [40] used a dynamic load balancing technique. Their method calculated the balance of each processor while sampling the cells. Data were redistributed between processors to achieve load balance.

Most of the presented solutions focused on how to improve blocks assignment to processors, which lead to better load balance. This is done either through a pre-processing step or at runtime. The work presented by [35] performed the camera transformation and had to use an estimation step to distribute the data dynamically.

While [40] achieved load balance, the cost of redistributing the data could be very expensive. This cost could be a bottleneck when the size of the data is large or if the camera is zoomed into a region of the data that belongs to one processor. This resulted in redistributing most of the data blocks in the camera view.

2.2.1.5 Hybrid Parallel Volume Rendering Solutions. Both image order and object order techniques have limitations and often can result in load imbalance. While several solutions have been proposed (Section 2.2.1.3, and 2.2.1.4) for both techniques to eliminate load imbalance, most of these solutions have additional costs such as a preprocessing step or redistribution of the data. Using a hybrid solution to overcome the limitations that both techniques have individually, and can reduce load imbalance at a lower cost.

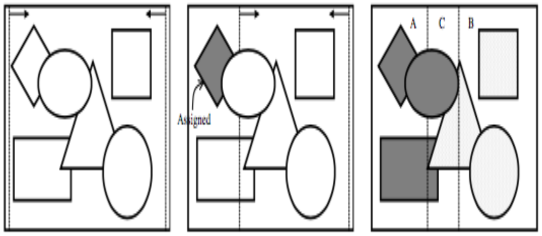
Montani et al. [41] presented a hybrid solution, where they used an image order distribution followed by an object order. In their work, nodes are divided into clusters, and the pixels are distributed among clusters using the image order technique. Each cluster loads the data contributing to its pixels, and data are distributed among nodes of the cluster using the object order technique. Their solution reduces the potential of load imbalance compared to traditional techniques, in addition to achieving data scalability. Load imbalance can still occur either at the clusters level or at the nodes level. At the clusters level, load imbalance can occur if some clusters were assigned an empty tile. At the nodes level, load imbalance can occur if some nodes of the cluster are assigned larger cells that need more work than others.

Childs et al. [42] presented another hybrid solution, where they used an object order distribution followed by an image order. In their solution, data were distributed among processors using the object order technique. Their solution began by categorizing cells into small and large cells, depending on the number of samples (see Section 2.2.1.2). Each processor was responsible for its own cells and classified them by comparing the number of samples with a given threshold. Next, each processor sampled small cells only. Then, pixels were distributed among processors using the image order technique. Depending on the pixels assignment, the algorithm exchanged two types of data: 1) samples that were generated from small cells, and 2) large cells that were not sampled. Next, each processor sampled the large cells contributing to its pixels. Then, samples from both sampling steps were composited generating a sub-image. Finally, sub-images were combined to produce the final image. As an extension for this algorithm, Binyahib et al. [43] presented a full evaluation of [42], where they compared the hybrid solution with traditional solutions. They also improved the original algorithm, where they reduced the memory and communication costs. In addition, their solution used hybrid parallelism to improve the performance and take advantage of many core architectures.

Samanta et al. [44] presented another hybrid solution that partitioned pixels into tiles and distributed cells into groups. Their algorithm used the camera view to determine visible cells. Next, the algorithm partitioned the visible region along the longest axis, assigning cells that are in the same screen space to the same processor. This is done by having two lines at the end of each side of the longest axis. The line moved into the opposite direction until there are N tiles, each containing N cell. Finally, each tile was assigned to a processor. Their solution achieved load balance by assigning N cells and N tiles to N processors. Figure 3 shows an example of the algorithm.

Garcia et al. [45] presented a hybrid algorithm, where they used an object order distribution followed by an image order. Their algorithm classified processors into clusters. Then data was distributed among different clusters using the object order technique. At each cluster, pixels were distributed among processors of the cluster using the image order technique. Next, communication happened between the different clusters to perform the image compositing step and produce the final image, thus reducing the communication cost. To reduce the memory requirement, their algorithm used an interleaved loading method. Each processor loaded every N_{th} row of the data, where N is the number of processors in the cluster. This meant that processors only had a partial data set to sample. Next, each processor used this sub-data to produce its part of the image, where interpolation was used for the missing rows. While this method reduced the memory cost, it came at the cost of image quality and accuracy. Increasing the number of processors per cluster had a direct impact on the final image accuracy. This method could be used to explore new data, but it would not be accurate enough to use for generating production images. In addition, load imbalance might still occur if the camera is focused on a region of the data that belongs to one cluster.

While the solution presented by [41] reduced the potential of load imbalance, this algorithm might not perform well in extreme camera conditions. For example, when the camera is inside the volume. The solution provided by [42, 43] performs better in these conditions, but it has additional communication cost in other camera positions such as when the camera is in the middle. The solution presented by [44] has an idle initialization time since all servers have to wait for the client to do the screen space transformation and then assign work to servers. While this algorithm might work on a small scale, it could perform poorly on a large scale. Finally, the solution presented by [45] reduces the potential of load imbalance and reduces the cost of the image compositing step. But load imbalance might still occur if some clusters have more work than others due to the camera view focus.



(a) Beginning (b) 1 object assigned (c) All objects assigned

Figure 3. Example execution of the hybrid partition algorithm [44].

2.2.1.6 Summary.

Table 1 shows a summary of the factors mentioned in Section 2.2.1.2 and the best configuration for each of these factors using image order and object order techniques. Each one of these factors impacts the choice of the technique, but these factors should be all considered when choosing a technique.

For example, [31], and [32] presented solutions to redistribute the workload to avoid load imbalance when using image order for the zoomed out case. While this could achieve good results when the size of the data is small, it could become very expensive when the data size increases. Another example is [35] and [40] solutions to reduce load imbalance for object order when the camera is inside the volume. While their solution reduced imbalance they added an additional cost of redistributing the data. Ma et al. [37] solution avoided this cost, but it could suffer from load imbalance if the data has an unusual mesh, where the cell sizes differ in a strange pattern.

As mentioned in Section 2.2.1.2 the performance depends on the number of samples and the number of cells per processor. Load imbalance occurs when there is an uneven distribution in one of them. The hybrid solutions combined both image order and object order to limit the imbalance in these two factors. Thus they can be viable alternatives to the two traditional techniques. While these solutions improve performance and have better results, they still have some limitations or additional costs.

2.2.1.7 Unstructured Data and Volume Rendering.

An unstructured mesh represents different cell sizes and sometimes different cell types in an arbitrary order. Figure 4 shows an example of structured and unstructured meshes. Unlike structured data, unstructured data does not have an implicit

Table 1. Factors impacting the performance of parallel volume rendering, and the best configuration for each of these factors using image order and object order techniques.

	Image Order	Object Order
Camera Position	zoom in	zoom out
Moving Camera View	No	Yes
Image Size	Large	Small
Data size	Small-Medium	Large

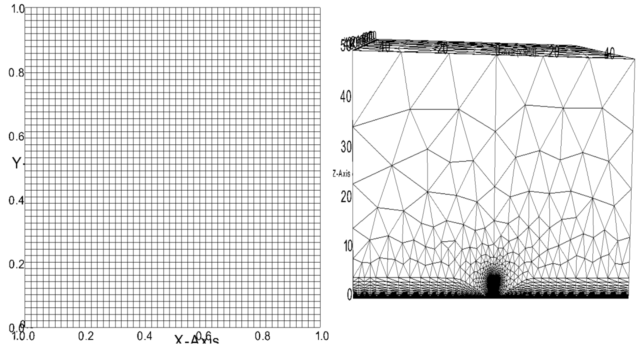


Figure 4. Example of (a) structured and (b) unstructured meshes.

indexing approach, and thus the cell connectivity information is not available. This increases the complexity of volume rendering.

Different solutions have been proposed to reduce this cost. Ma [46] presented an algorithm that computed the cell connectivity in a pre-processing step so it would not impact the performance while rendering. Each processor performed this step to acquire the cell connectivity information. In this step, the algorithm specified the external faces, which are faces that are not shared between cells. Next, the algorithm stored face-node, cell-face, and cell-node relationships in a hierarchical data structure. The algorithm excluded the cells that were outside the camera view. Then, each processor sampled its data. For each ray, it entered the volume from an external face, and the cell connectivity information was used to determine the next cell. A ray exited the volume when it intersected a second external face. Finally, the image compositing step was performed to exchange samples between processors and produce the final image.

Max et al. [47] proposed an algorithm that used slicing. Three slices were generated for each cell perpendicular to the X, Y, and Z axes. Depending on the camera view, one of these slices was used. While sampling, the values were computed using interpolation between the cell vertices. Next, the computed scalar values were used as 1D texture coordinates to obtain the color. Finally, the slices were rendered in back-to-front order, starting with slices that were furthest from the camera. The colors of these slices were composited to produce the final color.

Larsen et al. [48] presented an algorithm where cells were sampled in parallel using multi-threading. Cells were distributed among different processors. Each processor created a buffer that has the size of $Width \times Height \times NumberofSamplesperRay$. Each processor sampled its cells in parallel and samples were stored in the buffer. The index of each sample in the buffer was computed depending on its screen space coordinates (x, y, z). Finally, in the image compositing step, processors exchange samples, and samples of each ray were composited to produce the final color.

The solution presented by Ma [46] had the additional cost of the pre-processing step, which could become expensive when the data size is large. While Max et al. [47] algorithm did not have this cost, their algorithm might have a high cost at the compositing step. This is because their algorithm composited the slices in a back-to-front order, which means they cannot use the early termination technique, mentioned in Section 2.2.1.1. The algorithm introduced by Larsen et al. [48] could take advantage of the early termination techniques if the image compositing was done in front-to-back. But their algorithm can suffer from high memory cost if the size of the image ($Width \times Height$) is large and/or the number of samples is large.

2.2.2 Contouring.

One of the most used visualization techniques is iso-contours. An iso-contour displays a line or a surface representing a certain scalar value. This value is represented by an isoline in the case of 2D data or an isosurface in the case of 3D data. For example, displaying the isosurface of the density in a molecular simulation to represent the boundaries of atoms. There are different techniques for isosurface extraction; the most commonly used is Marching Cubes [49]. The marching cube method extracts a surface by computing triangles depending on a set of cases. Iso-contour extraction is composed of two steps: 1) the search step, and 2) the generation step. In the search step, the algorithm finds the cells containing the isovalue. In the generation step, the algorithm generate the isosurface triangles through interpolations of the cells scalar values. The computational cost of this method increases with the size of the data set. A parallel solution is therefore needed to process large data sets.

Out of core solutions have been proposed to handle large data sets. While these solutions are useful, they add additional I/O costs. Chiang et al. [50] presented an isosurface extraction algorithm that was an extension of a previous work [51]. The extension included parallelizing the I/O operations and isosurface extraction. Their work introduced a concept called meta-cells. Cells that are spatially near each other were grouped into a meta-cell. Their algorithm used a preprocessing step that partitioned the dataset into spatially coherent meta-cells. These meta-cells were similar in size. Thus the cost of reading these cells from memory is similar. Each meta-cell had two lists: a list contained the vertices information, and a list contained the information of the cells . For each vertex in the first list, the algorithm stored x, y, z, and a scalar value. For each cell in the second list, the algorithm stored pointers to the vertices of the cell. Using pointers allowed the algorithm to avoid storing each vertex more than once for each meta-cell. For each meta-cell, the algorithm computed meta-intervals, each meta-interval stored min and max values. Next, the algorithm computed a binary-blocked I/O (BBIO) interval tree, which is an indexing structure. The BBIO stored meta-intervals and the meta-cell ID for each interval; this ID is a pointer to the meta-cell. The algorithm stored the meta-cells and the BBIO on the disk. During run time, the algorithm used the BBIO to find the meta-cells that intersected with the isovalue. Next, the algorithm read meta-cells from disk one at a time and generated the isosurface triangles. In their algorithm, they used a self-scheduling technique [52], where one node acted as a client that assigned work to servers. The client scanned the BBIO and determined the active meta-cells. Next, the client maintained a queue of all active meta-cells. When servers had no more work, they sent a request to the client to be assigned more work. Each server read meta-cells from disk and computed isosurface triangles.

Another out of core solution was proposed by Zhang et al. [53]. Their algorithm maintained load balance by decomposing the data depending on their workloads. They used the contour spectrum [54] to get the workload information. The contour spectrum is an interface that provided a workload histogram for different isovalues. The algorithm reduced the I/O time by using a new model, where instead of having one disk that can be accessed by different processors, each processor has a local disk. Thus different processors could read data from their local disks in parallel. When a processor needed data from a remote disk, data were sent by the owner processor. For each local disk, the algorithm built an I/O-optimal interval tree [55] as an indexing structure. During run time, each processor searched its local disk for active cells and computed isosurface triangles.

Additional challenges are arising when extracting an isosurface on an Adaptive mesh refinement (AMR) data [56]. Different regions of simulation data need different resolutions depending on the importance of accuracy in that region. Adaptive mesh refinement (AMR) solves this by giving a finer mesh to regions of interest. AMR data is a hierarchy of axis-aligned rectilinear grids which is more memory efficient than using unstructured grid since it does not require storing connectivity information. While AMR reduces memory cost, it can create discontinuities at boundaries when transitioning between refinement levels, thus causing cracks in the resulting isosurface. One way to prevent the formation of these cracks is by creating transition regions between the different refinement levels [57]. Generating such transition region is difficult because of the difference of resolution between two grids and the hanging nodes (or T-junctions) caused by this difference. Hanging nodes are nodes found at the border between two grids but which only exist in the fine grid. Weber et al. [58] presented a solution that used dual grids [59] to remove these discontinuities. Their implementation mapped the grid from cell-centered to vertex-centered by using the cell centers as the vertices of the vertex-centered dual grid. This resulted in a gap between the coarse grid and the fine grids, which they solved by generating stitch cells between coarse and fine regions. Figure 9 shows a 2D dual grid before and after stitch cell generation. The approach used a case table to determine how to connect vertices to form suitable stitches. Performing isosurface extraction in parallel can lead to artifacts around the boundaries of the different data blocks. This happens because a processor does not necessarily own all the neighboring cells of its local cells. Instead, some neighboring cells can be owned by other processors. Thus their algorithm used ghost cells to avoid these artifacts. Since AMR data has different resolution levels, the algorithm decomposed data into boxes, where each box had one level only. Data was distributed among different processors and each processor performed the iso-surface extraction and generated stitch cells for its local data.

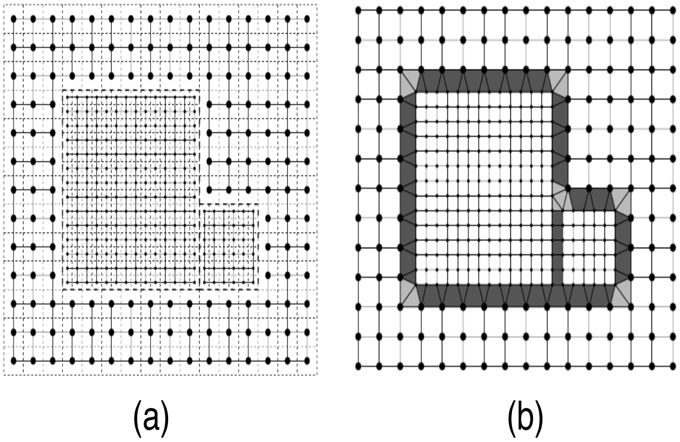


Figure 5. A Dual grid with three refinement levels. (a) Before stitch cell generation, the original AMR grids are drawn in dashed lines and the dual grids in solid lines. (b) After stitch cell generation [59].

The solution proposed by Chiang et al. [50] maintained load balance by using the self-scheduling technique (one client assigns the work to all the servers). However, this technique can become inefficient at large scale because many servers have to communicate with a single client. This might lead to having high communication cost. The solution proposed by Zhang et al. [53] used a pre-processing step to guarantee load balance across processors. Despite the addition of a pre-processing step, high communication cost could still happen because of block exchanges between processors which can be expensive for large data sets.

The solution proposed by Weber et al. [58] for AMR data is efficient, but it is dependent on the existence of ghost data. In cases where ghost data was not generated by the simulation code, it needs to be dynamically generated, which can increase the total execution time.

2.3 Supporting Infrastructure

In this section, we discuss different supporting algorithms that are used in parallel visualization.

2.3.1 Image Compositing. Image compositing is the final step of parallel volume rendering when using the object order technique (Section 2.2.1.4). The goal of this step is to order samples in the correct depth order to compute the final pixel color. Image compositing includes two operations: 1) communicating samples between processors, 2) compositing these samples to produce the color of the pixel. Image compositing is expensive and can become the bottleneck of the object order approach [34]. Thus several solutions have been proposed to reduce the cost of this step. In this section, we survey and compare these solutions.

2.3.1.1 Image Compositing Methods. There are three main image compositing methods: 1) direct send, 2) binary swap, and 3) radix-k.

The most straightforward method to implement is direct send [60], where all processors communicated with each other. In this method, image pixels were assigned to processors, where each processor was responsible for compositing a part of the image. Depending on this assignment, processors exchanged data. Figure 6 shows an example of a direct send compositing between four processors. While direct send is easy to implement it could be inefficient with a large number of processors since all processors are communicating with each other.

Another image compositing method is binary swap [36]. This method required the number of processors to be a power of two. In this method, the communication between processors happened in rounds. The algorithm performed $log_2(N)$ rounds, where N is the number of processors. Processors communicated in pairs, and each round the pairs were swapped. At each round, the size of the exchanged tiles was reduced by half. Figure 7 shows an example of a binary swap compositing between four processors. Binary swap reduced network congestion and had good scalability [34], but it had the limitation of requiring the number of processors to be a power of two. Thus an improved version, 2-3 swap, was implemented by Yu et al. [61] to overcome this limitation. Their algorithm worked with any number of processors and processors communicated in rounds. At each round, processors were divided into groups of size two and three, and processors in the same group communicated with each other.

This method had the flexibility in the number of processors while taking advantage of the efficiency of the binary swap. Another improved version of binary swap is 234 composite [62, 63]. Their solution used an elimination process named 3-2 and 2-1 [64] that was developed for optimizing reduction for a non-power of two number of processors. The 234 compositing method divided processors into groups of size three and four. For each round, a pair of processors exchanged half the image. At the end of a round, all processors of the same group have communicated and the result from each group is two halves of the image. The total number of half images produced from all groups is a power of two. A binary swap method is applied to collect these partial images into a full image.

Peterka et al. [65] proposed another image compositing method known as radix-k. Their method also performed communication in multiple rounds. At each round, it defined a group size k_i, where i is the current round. The multiplication of the group sizes of all rounds is equal to N, where N is the number of processors. For this algorithm, the product of all k_i must be equal to N. At each round, each processor was responsible for $1/k$ of the image.

Processors within a group communicated with each other using a direct send. Figure 8 shows an example of a radix-k compositing between six processors. This method avoided network congestion while providing the flexibility to work with any number of processors.

Moreland et al. [66] introduced a technique named telescoping to deal with non power of two number of processors. This technique grouped the largest power of two processors and defined it as the largest group. Then it took the largest power of two processors from the remaining processors and defined it as the second largest group. This process continued until all the processors have been assigned to a group. In each group, processors applied a compositing method, either binary swap or radix-k. Next, the smallest group sent its data to the second smallest group for compositing. The second smallest group did the compositing and sent the data to the third smallest group. This continued until all the data was sent to the largest group. They compared binary swap and radix-k using telescoping against the traditional methods, and their results showed overall improvement.

Direct send is flexible and easy to implement. While it has been used in several solutions, its performance can decrease when the number of processors is large due to the increase in the number of messages. Binary swap and radix-k solve this by allowing groups of processors to communicate at each round. Although this

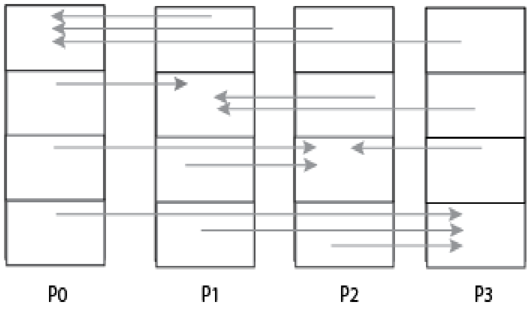


Figure 6. Image compositing using the Direct Send method between four processors.

reduces the communication cost, it introduces a synchronization overhead at the end of each round.

2.3.1.2 Image Compositing Optimization.

While the previous section focused on communication patterns for image compositing, in this section, we discuss optimization methods that have been presented for the compositing operation.

Active pixel encoding has been used to reduce the cost of image compositing. When using active pixel encoding, the bounding box and opacity information is used to mark inactive pixels. These pixels are removed to reduce the cost of communicating and compositing. Using this technique showed improvement in the performance in several solutions [67, 68, 69, 70, 71].

Load imbalance can increase the cost of image composing. This happens when a part of the image contains more samples; thus the processor that owns this part of the image has to do more work. Thus different solutions have been proposed to reduce load imbalance. One of the methods used is interlace [30, 70], where non-empty pixels are distributed among processors. The data pixels are rearranged so that all processors have a similar workload. While the traditional interlace technique has its advantage, it introduces an overhead at the final step

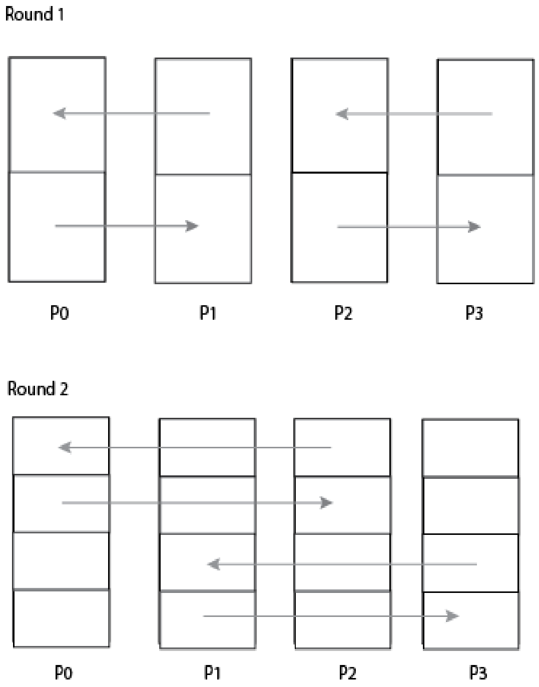


Figure 7. Image compositing using the Binary Swap method between four processors.

to arrange the pixels into their correct order and this overhead could be expensive when the image size increases. To reduce this cost, Moreland et al. [66] proposed an improvement. Their solution guaranteed the slices that are created during the data rearrangement are equal to the final image partitions created by the compositing method (binary swap or radix-k). Thus reducing the cost of pixels arrangement by avoiding extra copies which would have been necessary if the slices sizes did not match the final image partitions.

2.3.1.3 Image Compositing Comparative Studies. In this section, we discuss some of the papers that compared different image compositing methods.

Moreland et al. [66] compared the traditional binary swap method with different factorings of radix-k, where the group size varies. They used the Image

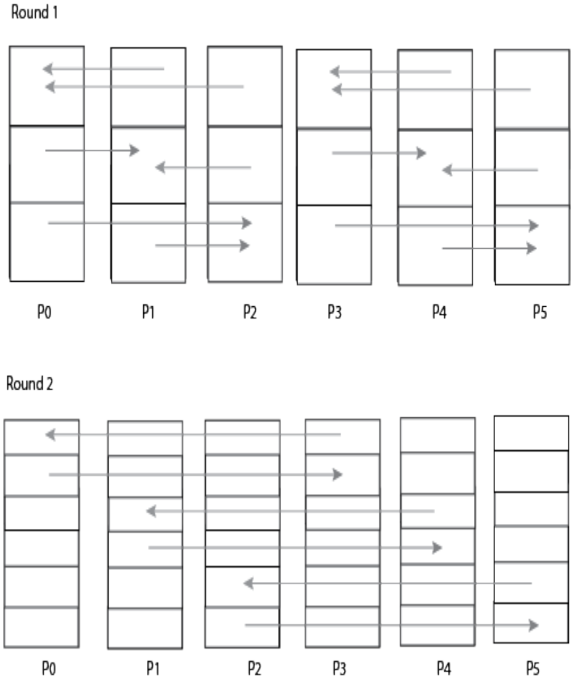


Figure 8. Image compositing using the Radix-k method between six processors and $k = [3, 2]$.

Composition Engine for Tiles (IceT) framework [72]. Their paper tested these methods at scale and added an improvement that was mentioned in the previous sections (Section 2.3.1.2). They compared binary swap and radix-k with these improvements against the traditional implementations and their results showed overall improvement.

Moreland [73] presented a paper where he compared different versions of the binary swap with the IceT compositor [72], which uses telescoping and radix-k. His paper focused on testing the performance when dealing with non-power of two number of processors. Variations of binary swap included 2-3 swap [61], 234 swap [62, 63], telescoping [66], a naive method, and a reminder method. The first three methods were discussed earlier in the section. The naive method finds the largest number of processors that is a power of two. Then, the remaining processors send their data to processors that are in the group and stays idle for the rest of the communication. The reminder method applies a 3-2 reduction to the remaining processors which is similar to the one mentioned in 234 compositing [62]. He ran each algorithm for multiple frames and different camera configurations. His experiments showed better performance for the telescoping and reminder methods, while the naive method performed poorly when dealing with a non-power of two number of processors. Finally, IceT showed better performance than all versions of binary swap.

2.3.1.4 Summary. There are two main factors impacting the performance of image compositing: 1) the number of processors, 2) the distribution of non-empty pixels, which is impacted by the camera position as mentioned in Section 2.2.1.2.

While different compositing and optimization methods have been proposed to improve the performance, sometimes paying the additional overheads introduced for these methods can be more expensive. When the number of processors is small enough, using direct send might result in better performance than using binary swap or radix-k. Since the number of processors is small, the probability of network congestion is low and thus it avoids the synchronization overhead introduced for more complex methods. As the number of processors increases, paying the cost of this overhead leads to better overall performance. If the distribution of non-empty pixels is dense in one region of the image (zoomed out camera position), this could lead to load imbalance. Thus using the optimization techniques mentioned in Section 2.3.1.2 and paying the additional cost can be necessary to improve the performance. Other cases show that simple solutions can be more efficient as well. According to [66] findings, the overhead of interlace could be larger than the gain when using a small number of processors. Another example is presented in [73], where the author showed that the reminder algorithm gives better performance than other more complicated methods.

2.3.2 Ghost Data. Parallel visualization algorithms usually distribute data among different processors, with each processor applying the algorithm on its sub-data. Different visualization algorithms depend on the values of neighboring cells, such as iso-contour extraction, and connected components. For example, in the case of isosurface extraction, interpolating the scalar value of a point depends on the scalar values of the neighboring cells. If a point is located on the boundaries of the sub-data, the result of the interpolation will be incomplete without considering the neighboring cells. Ghost data [74, 75] is used to allow parallelization of such algorithms. Ghost data is an extra set of cells added to the boundaries of the sub-data. These additional cells are usually only used for computations at the boundaries but are not taken into account during the rendering phase to avoid artifacts. For instance, in the case of iso surface extraction, ghost cells are used to correctly interpolate scalar values on the cells at the boundaries of the sub data, but the ghost cells themselves are not interpolated.

Different visualization tools support the use of ghost data, which has been used in previous solutions for different visualization algorithms such as isosurfaces [23, 22, 76, 58], particle advection [77], and connected components [78]. Ghost data is usually generated by the simulation code and most of visualization tools do not support the generation of the ghost data. Paraview provided a Data Decomposition (D3) filter [79] that generated ghost data by repartitioning the data. Patchett et al. [80] presented an algorithm to generate ghost data; this algorithm was integrated into the Visualization Toolkit (VTK) [19]. Each processor exchanged its external boundaries information with all other processors. Next, each processor compared its external boundaries with received external boundaries from all other processors. If the processor found an intersection, it sent the cells to the processor owning that boundary. Biddiscombe [81] presented an algorithm for generating ghost data, where he integrated the partitioning library Zoltan [82] with VTK and ParaView [23]. The algorithm provided the user with a selection of ghost cell generation options.

2.3.3 Metadata. Many visualization systems use a data flow framework. A data flow framework executes a pipeline of operations (or modules) with data being transmitted between modules. A pipeline usually applies different visualization algorithms known as filters. The optimization required to achieve good performance for visualization algorithms varies from one algorithm to another. It is important when optimizing to take into consideration the operations performed through the pipeline. For this reason, visualization frameworks use metadata, which is a brief description of the data that improves algorithms execution. There are different forms of metadata [83] including regions, and contracts.

Regions are a description of the spatial range of the whole data domain and the spatial bounds of different blocks. This information can be updated by the three pipeline passes [84] depending on the filters. For example, with a select operation, only a specific region of the data is needed. The pipeline updates the regions metadata so that only that part of the data is read.

Contract [14] is a data structure that provides optimization by allowing each filter to declare its impact. The data structure has data members that define constraints and optimizations. Each filter in the pipeline modifies this data member to make sure it contains its constraints and optimization requirements. Before performing any of the filters, the contract is passed to each filter in the pipeline starting from the last filter. After this process is done, filters are executed with the required optimizations. Contracts can be used to specify different parameters, such as identifying ghost data, cells to exclude, type of load balancing used by the framework, etc. Different visualization tools used contracts in post-hoc [22] and in situ [85, 86].

2.3.4 Delaunay tessellation.

N-body simulations such as cosmological or molecular dynamics simulations generate particles. However, it may be necessary to derive a mesh from these particles to better analyze and visualize certain properties. This is for instance, the case in cosmology simulation to analyze the density of dark matter [87, 88] Delaunay tessellation [89] is a geometric structure for creating a mesh from a set of points. Performing tessellation on large simulations is computationally expensive and must be performed in parallel.

Peterka et al. [87, 88] presented an algorithm that performed tessellation in parallel. Their algorithm distributed the data among different processors which then exchanged needed neighboring points. Each processor computed the tessellation using one of two libraries Qhull [90], or CGAL [91]. Then, each processor wrote the results to memory. To balance the number of points per block, a solution was proposed by Morozov et al. [92] where they used kD tree decomposition.

2.3.5 Out of Core.

Out of core algorithms [93, 94] (external-memory algorithms) have been used to allow visualization of large data that does not fit into the main memory. In out of core solutions, data is divided into pieces that can fit into main memory. An out of core algorithm reads and processes one piece of data at a time. This process is known as streaming. There are two paradigms of out of core solutions [94]: 1) batched computations, and 2) on-line computations. In the batched computation paradigm, there is no pre-processing step, and the entire data is streamed one piece at a time. In the on-line computation paradigm, a pre-processing step is performed, and the data is organized into a data structure to improve the search process. Using the on-line computation paradigm is effective for visualization since usually only a portion of the data contributes to the final result.

Different pre-processing techniques have been used to improve I/O efficiency. These techniques include meta-cells [50], and binary-blocked I/O interval tree (BBIO Tree) [51, 95].

The out of core model has been used in several visualization algorithms such as particle advection [96, 97, 98], and isosurfaces [50, 53]. It is also used by many visualization tools such as VisIt [22], VTK [19], Paraview [23], and the Insight Toolkit (ITK) [99].

2.3.6 Usage of Visualization Systems. Modern visualization tools such as Paraview or VisIt support three different modes. The first one is a client-server model, where the user runs a lightweight client on a local machine and connects to a server (supercomputer) that hosts the data. The computations are performed on the server and visualization is streamed back (geometry or images) to the client machine for display. The second mode executes the entire pipeline in batch without displaying the visualization and saves images on the supercomputer. Finally, the third mode is using the local machine of the scientist exclusively. Data is transferred from the supercomputer to a local machine to execute the visualization pipeline and explore data. Even though this mode might be convenient for the end user, it is often not practical anymore due to the extreme size of today's data sets which prevent moving data outside of the supercomputer. Additionally, a local machine or small cluster would not have the computational power and/or memory to process data in a timely manner.

2.3.7 Hybrid Parallelism. Hybrid parallelism refers to the use of both distributed- and shared-memory techniques. A distributed memory algorithm runs multiple tasks across multiple nodes in parallel and tasks communicate via the message passing interface (MPI) [100]. Multiple tasks can be running on the same node, usually one MPI task per core. A hybrid parallel algorithm run a fewer number of tasks per node (usually one per node) and use the remaining cores via threading using OpenMP [101], or POSIX [102]. Threads on the same node share the same memory, which allows for optimization. It is possible to take advantage of multicore CPUs with MPI only by running multiple MPI tasks per node. However, the threading programming model has proven to be more efficient. It requires less memory footprint and performs less inter-chip communication. Hybrid parallelism showed improved performance for volume rendering [103, 104], and particle advection [105, 106].

CHAPTER III

PARALLEL PARTICLE ADVECTION ALGORITHMS

Parts of the text in this chapter came from my area exam, which received editing suggestions from Hank Childs.

This chapter describes the foundations of particle advection (Section 3.1), the parallel particle advection algorithms studied in this dissertation (Section 3.2), and discusses other research done on parallel particle advection (Section 3.3).

3.1 Foundations

This section provides an overview of the particle advection technique. Advection is the process of moving a massless particle depending on a vector field. This results in an integral curve (IC), which represents the trajectory the particle travels in a sequence of advection steps from the seed location to the final particle location. Particle advection is a fundamental building block for many flow visualization algorithms [107, 108, 109, 110, 111, 112].

3.1.1 Integration Methods.

Integral curves can be calculated in an approximated form using numerical integration methods [113]. The complete IC is calculated on a sequence of advection steps until reaching the maximum number of steps or exiting the data. At each step, a part of the curve is computed between the previous particle location and the current. The vector field around the current location is used to determine the direction of the next location.

There are different methods to calculate the next location. The Euler method [113] is the simplest and least expensive method. It uses only the vector field of the current location to calculate the next location. Equation 3.1 shows the Euler method, where p_{i+1} is the next location of the particle, p_i is the current location of the particle, h is the length of the advection step, and $v(t_i, p_i)$ is the vector field value at the current location at the current time step. Runge Kutta (RK) [1] is a higher order method that uses Euler in its steps. There are different orders of the method; the most commonly used is the 4th order method referred to as RK4. Using RK4 produces more accurate results than Euler, but it is more expensive since it uses more points. Equation 3.2 shows the RK4 method, where p_{i+1} is the next location of the particle, p_i is the current location of the particle, h is the advection step, and $v(t_i, p_i)$ is the vector field value at the current location at the current time step. In both methods, as the advection step size decreases the accuracy of the trajectory increases, as well as the complexity. And as the total number of advection steps increases, the accuracy of the trajectory increases as well as the complexity.

$$p_{i+1} = p_i + h \times v(t_i\, p_i) \tag{3.1}$$

$$\begin{aligned} p_{i+1} = p_i + \frac{1}{6} \times h \times (k_1 + 2k_2 + 2k_3 + k4) \\ k1 = v(t_i\, p_i) \\ k2 = v(t_i + \frac{h}{2} + \, p_i + \frac{h}{2} \times k1) \\ k3 = v(t_i + \frac{h}{2} + \, p_i + \frac{h}{2} \times k2) \\ k4 = v(t_i + h + \, p_i + h \times k3) \end{aligned} \tag{3.2}$$

3.1.2 Parallelization Overview. Particle advection is computationally expensive, and this cost increases when the data size is large and exceeds the limits of a single machine, which leads to distributing the data across multiple nodes. To advect a particle, the algorithm needs to have the needed data block on the same node as the particle. There are two main parallelization

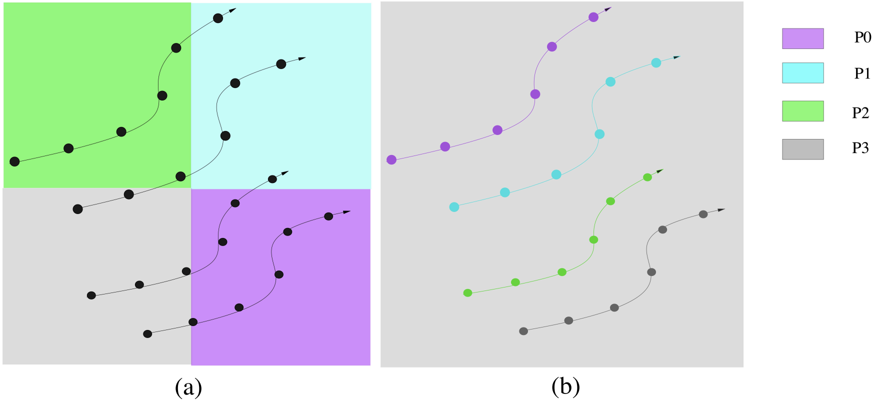


Figure 9. The distribution of work between 4 ranks using the two main parallel particle advection algorithms, (a) parallelize over data, (b) parallelize over particles.

techniques [105]: 1) parallelizing over data (see Section 3.2.1), and 2) parallelizing over particles (see Section 3.2.2). Figure 9 shows the distribution of work in the two algorithms. All additional parallel particle advection algorithms proposed to date are either an extension of one of these two or a hybrid between them. In this section, we start by discussing the challenges of parallel particle advection.

The efficiency of the parallelization algorithm can vary based on the characteristics of the workload and the computational resources available (number of nodes, memory per node, etc). We identified four main factors impacting the efficiency of these algorithms:

- **Data set size:** A given data set can be small enough to fit the main memory of a node or not. If the data set is small enough, it allows data to be replicated among nodes and favors the distribution of particles (parallelize over particles). As the size of the data increases, distributing data becomes necessary, thus parallelizing over data might lead to better performance. Another consideration is the number of cells per rank. Depending on the size of the memory, there is a limit on how many cells each rank can store.

The number of cells per block has to be small enough to fit into memory. But large enough to reduce the number of disk reads.

- **Total number of advection steps:** When the number of total advection steps needs to be computed is large, the computation complexity increases and thus distributing this complexity is important. If that number is small, then it is better to distribute the data (parallelize over data) to reduce the I/O cost.

- **Particles distribution:** Particles can be located in a region of the data (dense) or be more scattered (sparse). If the particle distribution is dense, only a subset of the data set will be required, reducing significantly the cost of I/O. This setup is more favorable to parallelize over particle because in the case of parallelizing over data only a small number of nodes would work. On the over hand, if the particles are spread out (sparse) and cover the whole data set, the cost of I/O will become more significant. In this case, parallelizing over data would be more favorable to limit the cost of I/O.

- **Number of MPI ranks:** Distributing the workload among multiple MPI ranks increases the amount of computational power and memory available. This can reduce the number of I/O operations necessary as more data can be stored in memory and the number of particles per rank is reduced. However, additional communications may also be required to better load balance the workload.

3.2 Studied Parallel Particle Advection Algorithms

In this dissertation we study four of the most used parallel particle advection algorithms, which are described in this section.

3.2.1 Parallelize-Over-Data Algorithm (POD).

Parallelize over data was introduced first by Sujudi and Haimes [114]. In this method, data is distributed between different nodes. Each node advects the particles located at its block until they exit the block or terminate. When a particle leaves the current data block, the particle is communicated to the node that owns the needed data block.

This technique reduces the cost of I/O which is more expensive than the cost of computation. While this technique performs well for uniform vector fields and sparse particles distribution, it can lead to load imbalance in other situations. This technique is sensitive to particles distribution and vector field complexity. Particles distribution can impact this method negatively in cases where the particles are located in a certain region of the data. Thus load imbalance might occur due to the unequal work distribution. In cases where the vector field is circular, the communication cost can increase. Examples of both cases are discussed in Section 3.1.2.

3.2.2 Parallelize-Over-Particles Algorithm (POP).

In this technique, particles are distributed across different nodes. Particles are sorted spatially before distributing them to different nodes to enhance spatial locality. Each node advects its particles and loads data blocks as needed. To minimize the cost of I/O, a node advects all particles that belong to the loaded block until the particles are on the boundaries of the block. This technique needs to cache blocks, and frequently uses the least-recently used (LRU) approach. If there is not enough space when a new block is loaded, the least recently used block is discarded. Each node terminates when all its active particles are terminated.

Camp et al., [115] used an extended memory hierarchy to reduce the cost of I/O. In their solution, data was stored in solid state drives (SSDs) and local hard drives instead of the file system. The algorithm treats SSDs as a cache where data blocks are loaded. Since the cache can hold a smaller amount of data than memory, blocks are removed in a LRU mechanism when exceeding the maximum specified number of blocks. When a data block is not found in cache, the algorithm checks local hard drives before accessing the file system. This extended hierarchy increased the size of the cache which leads to less disk access and thus reducing the I/O cost.

3.2.3 Work Requesting Algorithm (WOR). While the parallelize over particles algorithm ensures having equal number of particles accross different nodes, the workload might still be unbalanced. This is because particles might have different advection steps (some particles terminated early) or the I/O cost of some workloads are higher than others. To guarantee equal workload, different parallel particle advection algorithms used dynamic load balancing methods. One of the load balancing methods is based on a work requesting [116]. This algorithm is an extension of the parallelize over particles algorithm. Similarly, the algorithm starts by diving and distributing a set of seed points equally across multiple nodes. Each node advects its particles and load data on demand. In this algorithm, when a node has no more work, it requests particles from another node. The requesting node is called a thief and the other node is called a victim.

3.2.4 Master/Worker Algorithm (MW). Both parallelize over data and parallelize over particles have limitations as presented in Sections 3.2.1, and 3.2.2. Hybrid solutions have been proposed to address these limitations and maintain load balance while reducing additional costs.

Pugmire et al. [105] proposed a hybrid solution known as the master/worker. In their algorithm, nodes were divided into groups, where each group had a master. The algorithm partitioned the data statically and loaded data blocks on demand. The master distributed particles between the workers and monitored the workload to ensure load balance. When a node needed a data block, the master followed a set of rules to decide whether the worker should load the block or send the particle to another worker. Their algorithm showed better performance than both traditional parallelization techniques and has been used in the VisIt framework [22].

3.3 Other Parallel Particle Advection Algorithms

In this section, we survey other parallel particle advection solutions that we do not study in this dissertation.

3.3.1 Extensions to the Parallelize-Over-Data Algorithm.

Different solutions have been presented to avoid load imbalance that might occur when using the parallelize over data algorithm (see Section 3.1.2). Peterka et al. [117] presented a solution that used round-robin block assignment to guarantee that nodes are assigned blocks in different locations. Their solution eliminates the load imbalance that could occur in cases where particles are located in a certain region of the data. While their method can reduce load imbalance, it can also increase the communication cost.

Different solutions have used a pre-processing step to maintain load balance. Chen et al. [118] presented an algorithm that reduced communication cost. Their solution considered the particles distribution and the vector field while partitioning the data into blocks. Their method partitioned the data depending on the vector field direction, thus reducing I/O cost.

Another solution that considered vector field was presented by Yu et al. [119]. Their solution clustered data based on their vector field similarity. Next, the algorithm computed a workload estimation for each cluster. This estimation was used while distributing the data among nodes.

Nouanesengsy et al. [120] presented a method that also used a pre-processing step. Their algorithm used a pre-processing step to estimate the workload of each block using the advection of the initial particles. The results from the pre-processing step were used to distribute the work among nodes. Each node was assigned a percentage of the work of each block. Blocks are loaded to all nodes that share the workload of the block. Their solution maintained load balance and improved performance. While these solutions resulted in better load balance, they introduced a new cost which is the pre-processing step. This cost can become expensive when the data size is large.

While the round robin solution presented in [117] was simple, it did show good results. This solution is sensitive to particles distribution and vector field complexity. The algorithm did reduce the potential of load imbalance when the particle distribution is dense, but it might still occur if the data size is large and a small region of the data has all the particles. This will lead to a small group of nodes doing all the work If the vector field is complex, such as circular, the communication cost can become expensive, and if the vector field has a critical point, this will lead to load imbalance.

Using a pre-processing step to distribute the workload among nodes can improve performance. While these solutions [118, 119, 120] can lead to better load balance, the cost of the pre-processing step might be expensive leading to reduced overall performance. This cost increases when the data size is large, and most likely the pre-processing step has to be performed by different nodes, thus introducing additional communication cost. Paying the additional cost might not lead to improved overall performance especially when the number of particles is small.

3.3.2 Extensions to the Parallelize over Particles Algorithm.

Different solutions have been presented to avoid load imbalance that might occur when using the parallelize over particles algorithm (see Section 3.1.2). One solution used a work stealing approach [121]. In this approach, once a node is done advecting its particles it steals particles from another busy node. The node stealing the particles is called a thief, and the other node is called a victim. Each node stores its particles in a queue, the thief node transfer particles from the victim's queue. The most common approach to choose a victim is randomly [4]. Work stealing showed good results but it is difficult to implement.

Other solutions used k-d tree decomposition to balance the workload during run time. Morozov et al. [92] presented a solution that used k-d tree decomposition to redistribute workload. Their algorithm checked for active particles at regular time intervals. Active particles are divided into groups, where all groups have the same workload. Next, each node was assigned a group. While this method achieved load balance, it required access to the entire data set, which can increase the I/O and memory cost. Zhang et al. [77] proposed a solution to avoid this cost. Their algorithm assigned a data block with ghost layers to each node before run time. When the algorithm performed the particles decomposition, it considered the data blocks assignment. Thus each node received particles which were in its data block. Their results showed improved load balance while maintaining the cost of I/O.

Since this technique loads data on demand, the cost of I/O dominates most of the run time. Data prefetching [122, 123] has been used to reduce this cost. The idea of data prefetching is to load the next predicted needed data block while advecting the current particles to hide the I/O cost. Since the performance of this method depends on the accuracy of the prediction, the I/O access patterns are stored. Several solutions [96, 97, 98] have computed an access dependency graph to improve prediction accuracy. They performed a pre-processing step to compute the graph.

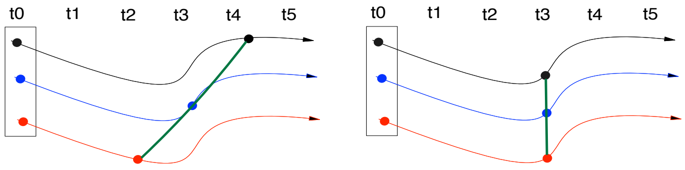
The most expensive step in parallelize over particles is I/O [124]. While several solutions [96, 97, 98] reduced this cost by using prediction to apply prefetching, they introduced additional costs and these cost increases as the data size and/or the number of particles increase. Camp et al., [115] reduced the I/O cost but the algorithm can still suffer from load imbalance if the advection steps vary between nodes. The solution suggested by [116] avoided load imbalance but at the cost of additional communications. While dynamic load balancing [92, 77] avoided the cost of pre-processing and it considered the change in the vector field, it added an additional cost of redistributing particles. This cost could increase when the number of particles is large.

3.3.3 Other Hybrid Algorithms.

Hybrid solutions have been proposed to address these limitations that both parallelize over data and parallelize over particles have individually. DStep [125] is a hybrid solution. They used a static round robin to assign the data blocks to nodes. Nodes were divided into groups, where each group had workers and had a communicator node (master). The algorithm stored particles in a queue, and the communicator assigned particles to nodes depending on their workloads. Nodes of the same group can communicate and send particles, and communicators of each group exchange particles between different groups. Their algorithm showed scalability and has been used in several implementations [126, 127, 128].

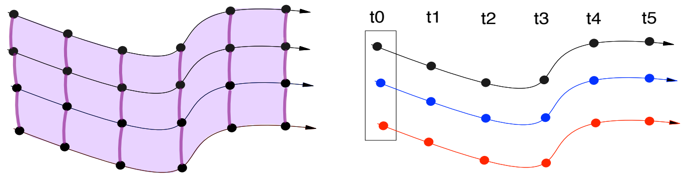
Lu et al. [129] presented another hybrid solution to compute stream surfaces (Section 3.3.4). A stream surface is a union of streamlines (particles trajectories) connected to form a surface. Their solution distributed data blocks among nodes. Next, particles were distributed among nodes in segments, where a segment is a part of the surface that is computed between two particles. Each nodes had a queue that stored the assigned segments. During run time, when a node needed a data block, it requested it from the node that owns the block. To make sure the load is balanced if particles of a segment diverge, the segment was divided into two segments and pushed to the node queue. When a node was idle, it acquired more work by stealing work from other nodes. Their results showed load balance and good scalability.

Hybrid solutions reduce load imbalance, but they are more complicated to implement, and they introduce additional costs. The solution presented by Pugmire et al. [105] does not require a pre-processing step, and it avoids redistributing the data. The algorithm showed better results than traditional techniques but it could still suffer from high I/O cost since it loads data on demand. Algorithms based on a master/worker design [105, 125] can perform poorly when the number of nodes is small, and the number of particles is large. This is because not all nodes are performing computation (advection). When the number of nodes is large, the communication between workers and masters can become a bottleneck, thus finding the correct group size can impact the performance. DStep [125] lowered the potential of the communication congestion between the workers and master by allowing nodes of one group to communicate directly. The solution proposed by Lu et al. [129] avoided this communication congestion, but it had the additional cost of communicating data blocks between nodes.



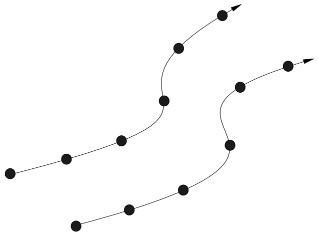
(d) A Streakline

(e) A Timeline



(b) A Stream Sufrace

(c) Pathlines



(a) Streamlines

Figure 10. Different flow visualization algorithms that use particle advection.

3.3.4 Flow Visualization Algorithms.

As mentioned previously, particle advection is used in many flow visualization algorithms. In this section, we give a brief description of some of these flow visualization algorithms, such as streamlines, pathlines, streaklines, timelines, and stream surfaces. A streamline [118, 106, 96, 105, 120, 117] is the trajectory of the particle from the seed location to the final location. Streamlines are the basis of other flow visualization algorithms. A pathline [119, 97, 98] is the trace of a particle through a period of time. Each pathline shows the moment of a certain particle through multiple time steps. A streakline is a line that connects the positions of different particles that passed a certain point. A timeline is a time that connects adjacent particles at a given time. A stream surface [130, 129] is a union of streamlines connected to form a surface. Figure 10 shows these different flow visualization algorithms. The most commonly used algorithms in scientific visualization are streamlines, pathlines, and stream surfaces.

As mentioned before, pathlines are traces of particles over time. This means that for each particle the algorithm is computing an additional value (three points for position and one for time), which increases the computational cost. In time varying data set, an additional challenge arises since particles might move from one block to another over time. Thus the change over time has to be taken into consideration. Yu et al. [119] presented a solution that used parallelize over data technique. Their algorithm considered time as a fourth dimension and performed a clustering based on the vector field similarity. Processors were assigned clusters depending on their workload, thus guaranteeing load balance over time.

The default setup for storing time varying data is to store each time step separately. Since a pathline algorithm computes the location of the next position in the next time step, the algorithm will need to access a different file with every integration step if parallelize over particles technique is used. This increases the I/O cost and might result in poor performance.

Chen et al. [97] presented an algorithm that reordered the storing of time varying flow data. Their algorithm used parallelize over particles technique and used data prefetching to load data blocks. The algorithm performed a preprocessing step to optimize the file layout and enhance the accuracy of prefetching. They divided the data into spatial blocks depending on their spatial locality. Next, particles that were in the same spatial block but in sequential time steps were grouped into a time block. In the pre-processing step, the algorithm computed an access dependency graph [96]. This graph was used to store time blocks and enhance data prefetching accuracy. Another solution that used access dependency graph to reduce I/O cost was presented by Chenet al. [98]. Their algorithm computed this graph in a pre-processing step, and grouped particles to the same block depending on their trajectories similarity. During run time, at each time step, nodes advected particles in groups. They are thus reducing the number of I/O operations.

Stream surfaces are computed using a front-advancing approach that was introduced by Hultquist [108] and used by other serial stream surfaces solutions [109, 110]. In this approach, the algorithm started by placing the seeding curve, which are the initial particles. Next, these particles are advected forming streamlines. An arc is created between adjacent pairs of streamlines; these arcs result in a stream surface. The computation of the surface depends on the advection of the particles at the front of the surface. New particles are inserted or deleted depending on the divergence or convergence of the surface. There are additional challenges when parallelizing stream surfaces. For example, when the particles in the front of the surface diverge, new particles needs to be added. This adds to the workload of the node owning that segment of the surface, which can lead to load imbalance. To reduce the potential of load imbalance,

Lu et al. [129] presented an efficient solution that used work stealing technique to balance the work between nodes. The algorithm is a hybrid between parallelize over data and parallelize over particles. Their solution divided the curve into segments that are distributed among nodes. Each node stored the segments in a queue and advected the particles in its segments. When the surface diverges and new particles are added, the algorithm formed new segments and inserted them to the node queue. When a node has no segments left, it requested segments from another node. Camp et al. [130] presented another solution for stream surfaces. However, their solution did not apply the front-advancing approach. Instead, their algorithm computed streamlines independently (regardless of the parallelization technique) and created the surface between these lines (triangulation) after advection. After the advection step, the algorithm performed an adaptive refinement check. If the distance between adjacent streamlines was larger than a given threshold, a new particle was inserted. This new workload was distributed between nodes (regardless of the parallelization technique) to perform the advection. Their algorithm reduced the potential load imbalance caused by the additional inserted particles.

Part II

Improving Individual Parallel Particle Advection Algorithms

This part of the dissertation discusses our improvements of current parallel particle advection algorithms.

CHAPTER IV

BEST PRACTICES AND IMPROVEMENTS TO THE PARALLEL ALGORITHMS

Parts of this chapter's text comes from comes from [8], which was a collaboration between David Pugmire (ORNL), Boyana Norris (UO), Hank Childs (UO), and myself. I am the first author of this publication and I wrote the majority of the paper, Hank Childs did significant editing, and Dave Pugmire did some review and editing. Boyana Norris provided the idea that started this work and provided feedback for the paper. I was the main implementer of the software for this study, but used a code base that David Pugmire contributed to. Hank Childs assisted in analyzing results.

This chapter describes the best practices for the individual parallel particle advection algorithms. We studied the different parallelization algorithms and their implementations and looked for the best practices presented in previous solutions in addition to possible improvements (Section 4.1). We also propose an improvement for one of the main parallel particle advection algorithms by integrating a scheduling method that has been used successfully in the HPC community (Section 4.2). Our results show that our proposed algorithm consistently improves the performance compared to traditional approaches.

4.1 Parallel Particle Advection Best Practices

In this section, we describe two best practices that have helped us optimize performance for our bake off study.

The first best practice is to incorporate shared-memory parallelism. We use the Many-Core Visualization Toolkit (VTK-m) [2, 3] library for shared-memory parallelism. It is a platform-portable library that provides efficient implementations of data-parallel primitives (DPPs) for different platforms. Using DPPs allows users to write a single DPP-based code for their algorithms that runs efficiently on different platforms, eliminating the need to rewrite the same code for different platform architecture.

The second best practice informs setting of the cache size. For all parallel particle advection algorithms that loads data on demand, blocks are removed from cache when the maximum number of blocks is exceeded. We adapted the settings presented by Camp et al. [115], while taking into account the data size and hardware differences between their study and ours. For our bake-off study, we allow 25 blocks per node, where each block has approximately two million cells.

4.2 A Lifeline-Based Approach for Work Requesting and Parallel Particle Advection

4.2.1 Motivation. One pitfall for POP (see Section 3.2.2) is that it suffers from idle time when some nodes finish their calculations before others. An important optimization for POP is to incorporate work requesting (see Section 3.2.3.) Work requesting is designed to minimize idle time — nodes that finish their calculations communicate with others nodes and request that they share some of their work. Work requesting incorporates an underlying scheduling method, and previous work has incorporated the Random Scheduling Method (RSM) [116, 131].

With this work, we introduce a new algorithm for work requesting parallel particle advection. Our improvement is to incorporate the Lifeline scheduling method (LSM). LSM is currently the high-performance computing community's preferred scheduling method for work requesting [5, 6, 7]. Our findings show that, for parallel particle advection, LSM is superior to RSM in all cases, and reduces inefficiency by significant amounts. Finally, since we discovered that RSM has some fundamental limitations for particle advection problems, we also introduce an extension to RSM to request work from multiple victims, which we refer to as RSM-N (N victims). That said, we find with our experiments that RSM-5 (5 victims) is also inferior to LSM.

4.2.2 Related Work.

Previous work requesting for particle advection solutions used random scheduling [4]. Several works in the high-performance computing community showed improved performance over random using a Lifeline-based scheduling method [5, 6, 7], which was introduced by Saraswat et al. [5]. In our algorithm, we replace the traditional random scheduling with the lifeline scheduling method.

The Lifeline approach begins similarly to the random scheduling method, in that the thief node attempts some number of random steals, w. But the lifeline algorithm differs in how to proceed if the first w steals all fail (i.e., did not result in work being returned from the victim, because the victim also has no work). Instead, the node consults its lifelines (i.e., a list of compute nodes) to ask for work. What differentiates a lifeline steal from a regular steal is that lifeline is then engaged on behalf of the thief to find work. Each of the lifelines will store the thief as an "incoming" lifeline. When those lifelines search for work themselves, they will share the work with the thief.

The key to the Lifeline approach is the "Lifeline graph," which directs a thief node to use specific nodes as lifelines.

The lifeline graph is a fully-connected directed graph, where graph vertices are compute nodes on the supercomputer and edges are lifelines. This graph must guarantee that there is a path from each node with work to all other nodes. The simplest way to create one lifeline for each node is to create a circular graph where the lifeline of the rank ID p is $(p + 1)\%N$, with N the number of ranks. This simple method is not acceptable in practice, though, since it will result in poor performance at scale. This is because the distance between two nodes is on average $\frac{N}{2}$ with N the total number of nodes. This means that requesting work to a victim would require on average $\frac{N}{2}$ communications, which is inefficient.

Instead, the Lifeline algorithm used a cyclic hypercubes graph to calculate the lifelines. This guarantees that the graph is connected, has a low diameter, and each vertex has a bound on the number of out edges. To calculate the lifelines of nodes, the user has to choose a base h and a power z, with the constraint $h^{z-1} < N \leq h^z$, where N is the number of compute nodes. Each node is represented as a number in base h with z digits, and has an outgoing edge to every node that is distance +1 from it in the Manhattan distance. Figure 11 shows a lifeline graph for of four nodes, with base $h = 2$, and power $z = 2$, each node has two lifelines. Full details on the method can be found in the paper by Saraswat et al.[5]

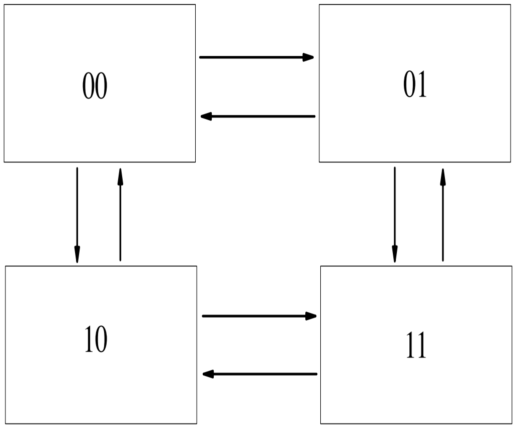


Figure 11. A lifeline graph of 4 nodes, with base = 2 and power = 2. Each node is represented in a base of 2 and has two lifelines. For each node, the outgoing arrows points to its lifelines.

4.2.3 Our Lifeline Algorithm.

This section describes our lifeline-based algorithm, as well as other algorithms we compare against. It is organized as follows:

- Section 4.2.3.1 describes foundational concepts.
- Section 4.2.3.2 describes two existing algorithms — POP and RSM.
- Section 4.2.3.3 describes our LSM algorithm.
- Section 4.2.3.4 describes an extension to RSM to include more victims (RSM-N); this algorithm allows us to evaluate lifeline better.

4.2.3.1 Foundational Algorithmic Concepts.

All four algorithms described share common elements. First, they each begin by dividing the set of P seed points over its N compute nodes, giving each node $\frac{P}{N}$ seed points to operate on. Each node then executes the same program, differing only in the seed points they begin with, and the algorithm completes when all particle trajectories are calculated. In the sections that follow, the pseudocode listed describes the program that runs identically on each node.

The pseudocode for our four algorithms use the following building blocks:

- Particle: a data structure that contains the particle to be advected through the flow field. This data structure contains the current location of the particle. It can optionally hold the previous locations of the particle (i.e., the trajectory) .
- ParticleArray: a data structure that contains an array of Particles.
- ArrayOfParticleArrays: a data structure that contains an array of ParticleArrays. For example, an ArrayOfParticleArrays with 10 entries would

contain 10 ParticleArrays, with each of the 10 ParticleArrays containing a varying number of particles.

- SortParticleByBlock(): a function that sorts Particles by the ID of the block that contains the particles. This generates an ArrayOfParticleArrays where the ParticleArray at index i contains the Particles that lie within block i.
- ObtainBlock(): a function that determines the needed block and reads it from cache or disk. The function first checks if the block is already available in cache. If not, it loads the necessary block and places it in cache. The size of the cache changes depending on the size of the data.
- Advect(): a function that advects the Particles of a ParticleArray until they exit the current block or terminate. This function returns a 2-tuple — the first element is a ParticleArray containing completed Particles and the second element is a ParticleArray containing Particles that exited the current data block.
- CheckForIncomingMessages(): a function that checks for incoming messages from other nodes. These messages can be work requests from other nodes or notifications of particle terminations.
- SendWork(): a function that sends half of its workload to the thief if it has any work, or sends back a “no work” message.
- RequestWork(): a function that requests work from another node.

4.2.3.2 Existing Algorithms. This section describes two existing algorithms used as comparators for our study: POP and Work Requesting using the RSM.

Algorithm 1 Pseudocode for the Parallelize-Over-Particles algorithm (POP).

```
function POP-Advect(ParticleArray pv)
    $keepGoing \leftarrow true$
    $ArrayOfParticleArrays\ \ pva[NUMBLOCKS]$
    $pva \leftarrow SortParticlesByBlock(pv)$
    $allCompletedParticles \leftarrow \emptyset$
    while keepGoing do
        $contParticles \leftarrow \emptyset$
        for i in NUMBLOCKS do
            if pva[i].size() > 0 then
                $Block\ \ b \leftarrow ObtainBlock(i)$
                $ParticleArray\ \ completed,\ \ continuing$
                $(completed, continuing) \leftarrow Advect(pva[i], b)$
                $allCompletedParticles\ \ += \ \ completed$
                $contParticles\ \ += \ \ continuing$
            end if
        end for
        if contParticles.size() > 0 then
            $pva \leftarrow SortParticlesByBlock(contParticles)$
        else
            $keepGoing \leftarrow false$
        end if
    end while
end function
```

Parallelize-Over-Particles. Algorithm 1 shows the pseudocode for POP. The algorithm starts by sorting particles by block. Then it reads the needed data blocks either from cache or disk. Next, the algorithm advects the particles located in the current data block until they terminate or exit the current block. When particles exit their current data blocks, they are stored in an array to be processed in the next iteration.

Even though the algorithm divides seeds equally between nodes, it does not guarantee an equal workload on each node. That is because nodes might load different number of blocks or have different number of advection steps, due to the nature of the vector field and placement of assigned seeds. For example, if the vector field has critical points attracting the particles toward them, the workload of the node depends on the placement of its assigned seeds. Nodes that have seeds located near the critical points will need fewer block than particles that are far from the critical point.

Work Requesting using the Random Scheduling Method. Algorithm 2 shows the pseudocode of RSM. The algorithm begins with each node executing the POP algorithm as described in Section 4.2.3.2.

The algorithm is different, however, in how it proceeds when a node finishes its work. In this case, it sends a work request to another node. Again, the node stealing the particles is referred to as thief and the other node is referred to as victim. RSM chooses a victim randomly [4]. If the victim has work, it sends half of its workload to the thief. Otherwise, it sends a "no work" message to the thief. In that case, the thief selects another victim randomly.

To optimize I/O, the algorithm sorts particles by block before sending work. This reduces the number of blocks that need to be accessed.

4.2.3.3 Our Lifeline-Based Algorithm. This section describes our particle advection Lifeline-Based algorithm. Algorithm 3 shows the pseudocode of LSM. LSM shares most of the steps of RSM, with the main difference between them being the scheduling method.

In this algorithm, the thief performs w random steals, where the victims are chosen randomly, and w is a user specified parameter. If no work is found after w attempts, the thief requests work from its lifelines. The lifelines are computed using a lifeline graph following the rules mentioned in Section 4.2.2. If the victim does not have any work, it requests work for its lifelines recursively. After the thief requests the work from its lifelines, it remains idle.

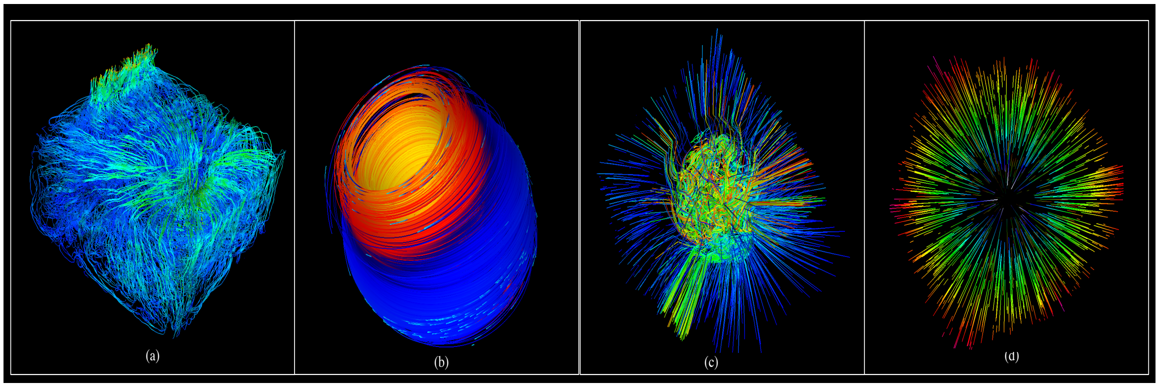


Figure 12. Streamlines visualization for the four data sets: (a) Fishtank, (b) Fusion, (c) Astro, (d) RadialExpansion.

When a node receives work, it checks for incoming lifelines; if it has any, then it sends work.

Similar to RSM, the algorithm sorts its particles by block before dividing the workload among lifelines, to reduce I/O cost.

The number of lifelines for each node impacts the performance of LSM. If the number of lifelines is small, it might lead to a higher idle time. On the other hand, if the number of lifelines is large, it might increase the communication cost.

4.2.3.4 RSM-N: Extending RSM For Multiple Victims.

For evaluation purposes, we also made a straightforward extension to the RSM algorithm, namely, to request work from multiple victims.

To conduct a fair comparison between the two scheduling methods, we adapted RSM to allow the thief to request work from the same number of victims as LSM. If no work is found, the thief chooses a new group of random victims.

4.2.4 Experiments. This section describes the details, which compares the four algorithms described in Section 4.2.3: POP, RSM, RSM-N, and LSM. The additional factors considered in this study are described in the following subsection.

4.2.4.1 Algorithm Comparison Factors.

Our study is composed of seven phases. The first phase considers one workload in depth, comparing the four algorithms. In the other six phases, one of six factors is varied, while holding the other five constant. The six factors are:

- Data set (4 options)
- Number of particles (4 options)
- Maximum advection steps (i.e., duration of particle) (4 options)
- Number of blocks (5 options)
- Number of cells per block (3 options)
- Number of MPI tasks (3 options)

In total, we considered 23 (= 4 + 4 + 4 + 5 + 3 + 3) configurations. We tested each configuration with all four algorithms, meaning 92 experiments overall.

Data Set. Since the complexity of the vector field impacts the performance, we test the performance of our algorithms on different data sets that broadly represent typical application scenarios. The four data sets used in this study are:

- The Fishtank data set is a thermal hydraulics simulation using the NEK5000 [132] code. In this particular simulation, twin inlets pump water of differing temperatures into a box. The mixing behavior and temperature of the water at the outlet of the box are of interest. The vector field captures the fluid flow within the box.
- The Fusion data set is a magnetically confined plasma in a tokamak device. The simulation was performed using the NIMROD [133] code. The vector field in this example is of the magnetic field that exists inside the plasma that

  is a result of the magnets in the tokamak device as well as the motion of the particles within the plasma itself.

- The Astro data set is the magnetic field surrounding a solar core collapse resulting in a supernova. This simulation was performed with the GenASiS [134] code, a multi-physics code for astrophysical systems involving nuclear matter.
- The RadialExpansion data set is an artificial dataset, where the vector for each point is measured by the distance from the location of the point to the center. The dataset was created to test the behavior of the four algorithms in cases where the load is highly imbalanced.

The four data sets are 3D steady data sets that were refined to a 1024^3 grid.

Number of Particles. With this factor, we consider the impact of the number of particles on the inefficiency of the four algorithms. Four amounts of particles are considered: 10K, 100K, 1M, and 10M.

Maximum Advection Steps. With this factor, we consider the impact of the advection steps on the inefficiency of the four algorithms. Four amounts of advection steps are considered: 100, 1K, 10K, and 100K.

Number of Blocks. With this factor, we consider the impact of the number of blocks on the inefficiency of the four algorithms. Five numbers of blocks are considered: 64, 128, 512, 1024, and 2048. For all five configurations, the total data size is 1024^3.

Number of Cells per Block. With this factor, we consider the impact of the block size (i.e., the number of cells per block) on the inefficiency of the four algorithms. Three sizes of blocks are considered: 64^3, 128^3, and 512^3. For each of these tests, we used 512 blocks.

Number of MPI Tasks. With this factor, we consider the scalability of our algorithms. This includes increasing the number of MPI tasks, as well as the data size and the number of particles. Three levels of concurrency are considered:

- Test 1: 32 MPI tasks, 5 lifelines, 1M particles, and 512 blocks.
- Test 2: 128 MPI tasks, 7 lifelines 4M particles, and 2048 blocks.
- Test 3: 512 MPI tasks, 9 lifelines 16M particles, and 8192 blocks.

4.2.4.2 Hardware Used. The study was run on Cori at Lawrence Berkeley National Laboratory's NERSC facility. It contains 2,388 Intel Xeon "Haswell" processor nodes. There are 32 cores per node, and each core supports 2 hyper-threads and 128 GB of memory per node.

4.2.4.3 Algorithms Configuration. RSM-N and LSM request work from multiple victims at each request. LSM calculates the number of victims using the equation in Section 4.2.2. We used that equation to compute the lifeline graph with a base equals to 2. All our tests except for the last phase use 32 MPI ranks, and thus the number of lifelines is equal 5. For the RSM-N algorithm, we used the same number of victims as LSM (i.e., RSM-5).

4.2.4.4 Performance Measurement. For each phase, we measure the work time (including I/O, advection, etc.), idle time, and total time. From these measurements, we can derive the inefficiency of the algorithm. Inefficiency affects the performance because the execution time is determined by the time of the

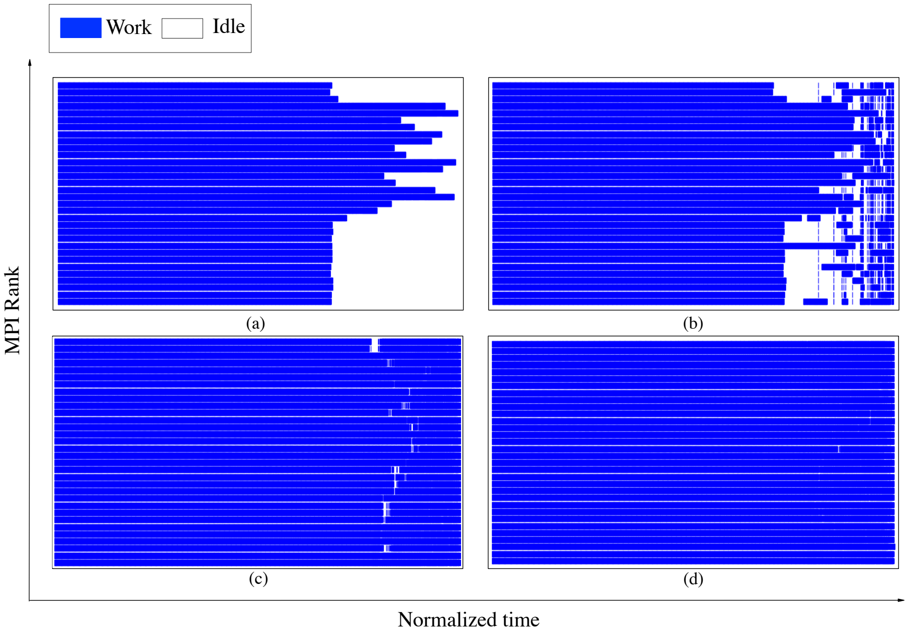


Figure 13. Performance of the four algorithms (a) POP, (b) RSM, (c) RSM-5, (d) LSM, using 32 MPI tasks to advect 1 million particles for 10 thousands steps (base case).

slowest processor. We define inefficiency with the following equation:

$$\text{Inefficiency} = \frac{\text{Idle}_{\text{time}}}{\text{Total}_{\text{time}}}$$

The idle time in our tests only measures the time that a node spends waiting for other nodes to finish their work. It does not include the time spent performing redundant I/O operations, which is more likely to happen with RSM, RSM-N, and LSM. Further, as the number of steal increases, it is more likely to perform redundant I/O. For this reason, we include in our results both total time and inefficiency. The goal of reducing inefficiency is to reduce the total execution time. It is important to make sure that the additional I/O and communication operations done to reduce inefficiency do not lead to a higher total execution time.

4.2.5 Results.

In this section, we present the results of our study.

Table 2. Comparing the performance of the four algorithms in terms of total execution time, the time for the individual routines, the idle time, and the inefficiency. The initialization time measures the time to initialize variables and generate initial seeds. The I/O time measures the time to read data blocks from disk or cache. The advection time measures the time to advect particles and to process the advection results (e.g., terminate). The communication time measures the time to request or send particles to other nodes and to inform other nodes of termination. The sorting time measures the time to sort particles by block after each round (line 18 in Pseudocode 1). The idle time measures the time where a node is waiting for other nodes to finish or send work. The inefficiency measures the percentage of execution time spent in idle and is computed as defined in Section 4.2.4.4.

	POP	RSM	RSM-5	LSM
Total time	131s	117s	111s	107s
Initialization time	1.68s	1.60s	1.55s	1.54s
IO time	14.8s	15.3s	22.9s	20.9s
Advection time	78.4s	75.0s	74.9s	73.6s
Communication time	$2.1e^{-4}$s	$7e^{-3}$s	0.15s	0.04s
Sorting time	9.21s	9.05s	8.80s	8.61s
Idle time	26.5s	16.3s	2.8s	1.7s
Inefficiency	0.20	0.14	0.03	0.02

4.2.5.1 Phase 1: Base Case. In this phase, we compare the performance of the four algorithms, using the following configuration:

- Data set: Fishtank
- Number of particles: 1M
- Maximum advection steps: 10K
- Number of blocks: 512
- Number of cells per block: 128^3
- Number of MPI tasks: 32 (512 cores)

The results of this phase are presented in Table 2. The results show a significant drop in inefficiency from 20% (POP) to 2% (LSM).

POP has the highest inefficiency, with 20% of the total execution time is spent idle. This is due to the load imbalance between nodes, which can be seen in Figure 13. Although the nodes have the same number of particles, their workload varies, see Section 4.2.3.2 for more discussion.

Using RSM reduces the inefficiency by a factor of 1.4. While this reduction improves the performance, idle time still takes 14% of the total execution time since thieves request work from one victim at a time.

Using multiple victims in RSM-5 reduces the inefficiency by a factor of 6.6 over POP, and a factor of 4.6 over RSM. This is because sending multiple requests at once allows the thief to receive work faster, and therefore reduces idle time.

LSM reduces the inefficiency by a factor of 10 over POP, a factor of 7 over RSM, and a factor 1.5 over RSM-5. LSM reduces the inefficiency compared to RSM-5 because the cyclic feature of the lifeline graph guarantees to always have a path from an idle node to a busy node.

Table 3. Comparing the number of advection steps and I/O operations between the four algorithms.

	POP	RSM	RSM-5	LSM
Total advection steps	9.97B	9.97B	9.97B	9.97B
Min advection steps	300M	231M	222M	230M
Max advection steps	312M	373M	383M	380M
Total # disk reads	3750	4081	5491	5381
Min # disk reads	67	87	126	128
Max # disk reads	256	221	216	198
Total # cache reads	2925	3009	3545	3587
Min # cache reads	1	4	30	28
Max # cache reads	244	223	195	207

I/O cost varies between the four algorithms. POP has the lowest I/O cost of all four algorithms, and it has the lowest number of I/O operations (disk and cache) as presented in Table 3. RSM has a higher I/O cost and I/O operations than POP. This increase is because particles are communicated between nodes and new data blocks are needed. RSM-5 and LSM have a higher I/O cost and I/O operations than RSM and POP. That is because more particles are communicated between nodes.

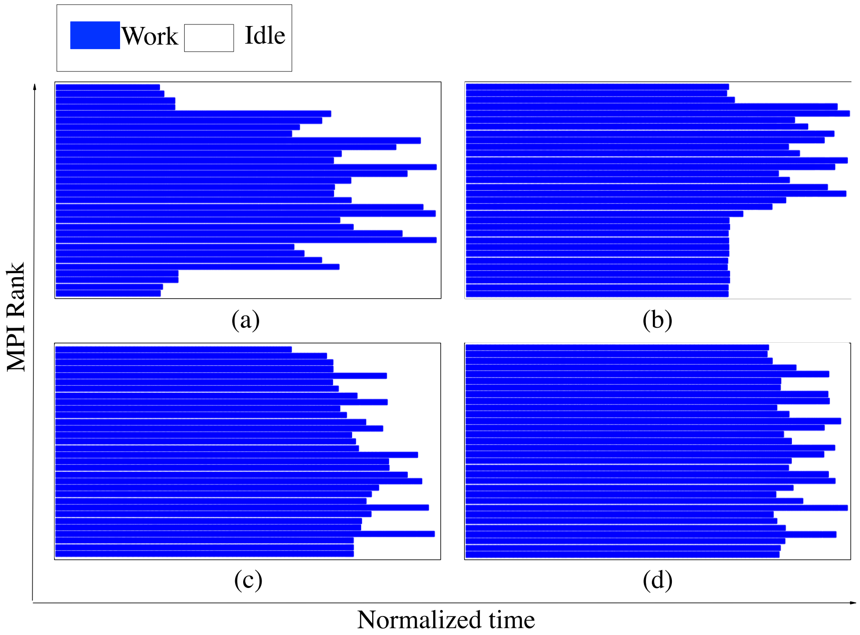
The advection time varies between the four algorithms, even though they are advecting the same number of particles. LSM does a better job balancing the workload, which leads to better usage of threads. On the other hand, when using POP the workload is not balanced, which leads to underused threads.

The communication cost varies between the three work requesting algorithms (RSM, RSM-5, LSM) because of the difference in their communication pattern. RSM has the lowest communication cost between the three algorithms since thieves communicate with one victim at a time. This results in a lower communication time at the cost of a higher idle time. Both RSM-5 and LSM communicate with the same number of victims at a single request. However, RSM-5 has a higher communication cost than LSM. This is because, in case of failure to receive work, LSM relies on its lifelines to receive work. RSM-5, on the other hand, needs to perform another request to 5 new victims until it receives work.

Even though the inefficiency is improving by a factor of 10, the total time is only improving by 20%. This is because there is a maximum improvement possible when improving a part of the program. This improvement is limited by the time needed to perform advection steps.

4.2.5.2 Phase 2: Data Sets. In this phase, we vary the data set using the following configuration:

- Number of particles: 1M



(a) (b) (c) (d)

Figure 14. Performance of the POP algorithm on the four data sets (a) RadialExpansion, (b) Fishtank, (c) Astro, (d) Fusion, using 32 MPI tasks to advect 1 million particles for 10 thousands steps.

- Maximum advection steps: 10K
- Number of blocks: 512
- Number of cells per block: 128^3
- Number of MPI tasks: 32 (512 cores)

Table 4. Comparing the inefficiency and time of the four algorithms when varying the data sets.

Data set		POP	RSM	RSM-5	LSM
Fishtank:	Inefficiency	0.20	0.14	0.03	0.02
	Total time	131s	117s	111s	107s
Fusion:	Inefficiency	0.12	0.09	0.02	0.01
	Total time	115s	109s	102s	101s
Astro:	Inefficiency	0.18	0.12	0.03	0.02
	Total time	126s	120s	103s	102s
RadialExpansion:	Inefficiency	0.33	0.27	0.04	0.03
	Total time	74.1s	73.5s	60.8s	54.9s

The results of this phase are presented in Table 4. The table shows that LSM reduces the inefficiency for all four data sets and maintains low inefficiency ratios for all cases (0.01-0.03).

For the RadialExpansion data set, the POP algorithm has a high inefficiency ratio (30%). This is because the workload is highly imbalanced, as can be seen in Figure 14 (a). Since vectors are moving from the center toward the boundaries of the box, nodes that are responsible for particles located in the center of the box have a higher workload. RSM reduces the inefficiency by only a modest factor of 1.2 over POP. RSM-5 and LSM, however, reduce the inefficiency over POP by a factor of 8.2 and 11, respectively.

For the Fishtank data set, the POP algorithm also has a high inefficiency ratio (20%). This is because the workload is highly imbalanced, as can be seen in Figure 14 (b). The velocity field is moving toward a sink at one end of the box. Nodes that are responsible for particles that are located on the opposite side of the box have more workload. RSM reduces the inefficiency by a factor of 1.4 over POP. RSM-5 and LSM reduce the inefficiency over POP by a factor of 6.6 and 10, respectively.

For the Astro and Fusion data sets, the POP algorithm has a lower inefficiency ratio compared to the previous data sets: 18% for the Astro data set and 12% for Fusion data set. The workload of the POP algorithm is less imbalanced for these two data sets compared to the other two (Figure 14 (c) and (d)). This is because both data sets have a more uniform vector field. However, using RSM-5 and LSM still reduce the inefficiency significantly.

4.2.5.3 Phase 3: Number of Particles. In this phase, we vary the number of particles, using the following configuration:

- Data set: Fishtank
- Maximum advection steps: 10K
- Number of blocks: 512
- Number of cells per block: 128^3
- Number of MPI tasks: 32 (512 cores)

Table 5. Comparing the inefficiency and time of the four algorithms when varying the number of particles.

Number of Particles		POP	RSM	RSM-5	LSM
10K:	Inefficiency	0.25	0.23	0.17	0.11
	Total time	15.4s	14.2s	13.0s	12.1s
100K:	Inefficiency	0.28	0.20	0.15	0.07
	Total time	28.6s	26.6s	23.4s	22.6s
1M:	Inefficiency	0.20	0.14	0.03	0.02
	Total time	131s	117s	111s	107s
10M:	Inefficiency	0.44	0.27	0.04	0.03
	Total time	1278s	921s	501s	486s

The results of this phase are presented in Table 5. The table shows that LSM reduces the inefficiency in all cases, with the highest improvement being a reduction of 14.6 over POP, for 10M particles.

The table shows that the inefficiency of POP is not directly correlated to the number of particles. At each test, the number of particles is increased by a factor of 10, but the inefficiency does not change within the same ratio and in one case it drops (in the case of 1M). This is because the inefficiency change is not dependent on the total number of particles, but rather on the workload distribution per node (Section 4.2.3.2). On the other hand, both RSM-5 and LSM are able to reduce the inefficiency consistently.

4.2.5.4 Phase 4: Number of Steps. In this phase, we vary the number of advection steps, using the following configuration:

- Data set: Fishtank
- Number of particles: 1M
- Number of blocks: 512
- Number of cells per block: 128^3
- Number of MPI tasks: 32 (512 cores)

Table 6. Comparing the inefficiency and time of the four algorithms when varying the durations of particles (maximum advection steps).

Advection Steps		POP	RSM	RSM-5	LSM
100:	Inefficiency	0.07	0.07	0.05	0.03
	Total time	20.3s	20.4s	20.2s	18.0s
1K:	Inefficiency	0.20	0.11	0.07	0.04
	Total time	33.7s	31.4s	29.3s	27.1s
10K:	Inefficiency	0.20	0.14	0.03	0.02
	Total time	131s	117s	111s	107s
100K:	Inefficiency	0.19	0.12	0.02	0.01
	Total time	1080s	981s	877s	791s

The results of this phase are presented in Table 6. LSM reduces the inefficiency in all cases, with a reduction of a factor of 19 in the case of 100K steps.

For the case of 100 advection steps, POP has an inefficiency of 7%. This is because the number of advection steps is small, making it less likely for particles to travel across multiple blocks or exit the domain. This reduces the probability of imbalance between nodes. RSM has the same inefficiency ratio as POP. This is because nodes have small workloads. Consequently, it is more difficult for a thief to find a victim with work available.

However, both RSM-5 and LSM reduce the inefficiency over POP by a factor of 1.4 and 2.3, respectively. As both methods are requesting work from five victims at a time, they are more likely to find work and therefore reduce the inefficiency.

Increasing the number of advection steps from 100 to 1k increases the inefficiency of POP to 20%. The table shows that the inefficiency of POP is not directly correlated to the number of advection steps but to the distribution of workload. On the other hand, both RSM-5 and LSM are able to consistently reduce the inefficiency down to 2% and 1%, respectively.

4.2.5.5 Phase 5: Number of Blocks. In this phase, we vary the number of blocks. That said, the overall data size remains constant through all tests (1024^3). The other factors for this configuration are the following:

- Data set: Fishtank
- Number of particles: 1M
- Maximum advection steps: 10K
- Number of MPI tasks: 32 (512 cores)

Table 7. Comparing the inefficiency and time of the four algorithms when varying the number of blocks.

Number of Blocks		POP	RSM	RSM-5	LSM
64:	Inefficiency	0.32	0.24	0.19	0.06
	Total time	89.1s	83.1s	82.7s	80.0s
128:	Inefficiency	0.25	0.21	0.09	0.04
	Total time	71.2s	68.9s	67.3s	65.1s
512:	Inefficiency	0.20	0.14	0.03	0.02
	Total time	131s	117s	111s	107s
1024:	Inefficiency	0.24	0.16	0.02	0.01
	Total time	168s	151s	131s	122s
2048:	Inefficiency	0.30	0.16	0.01	0.01
	Total time	257s	219s	174s	172s

The results of this phase are presented in Table 7. It can be seen that LSM reduces inefficiency including by a factor of 30 for the largest number of blocks.

The results show that the inefficiency of the POP algorithm is not directly impacted by the change in the number of blocks. This is because, in the POP algorithm, nodes perform their computation independently without communicating with other nodes. Therefore loading more blocks on each node does not affect the overall load balance of the POP.

The inefficiency of the three other algorithms (RSM, RSM-5, LSM) reduces as the number of blocks increases. This is because these algorithms respond faster to work requests as the size of the block reduces. These algorithms run an iterative loop. At the end of each iteration, the nodes check for work requests and send the appropriate responses (Algorithm 2 line 17). Reading smaller blocks reduces the time spent in I/O, reducing the time between requests.

4.2.5.6 Phase 6: Cells per Block. In this phase, we fix the number of blocks to 512 and vary the size of blocks (i.e., data size), using the following configuration:

- Data set: Fishtank
- Number of particles: 1M
- Maximum advection steps: 10K
- Number of blocks: 512
- Number of MPI tasks: 32 (512 cores)

The results of this phase are presented in Table 8. The table shows that LSM reduces the inefficiency for the different sizes of blocks, with a factor of 22 for the smallest size.

Table 8. Comparing the inefficiency and time of the four algorithms when varying the number of cells per block with 512 blocks in total.

Cells per Block		POP	RSM	RSM-5	LSM
64^3:	Inefficiency	0.22	0.13	0.02	0.01
	Total time	131s	123s	100s	95.7s
128^3:	Inefficiency	0.20	0.14	0.03	0.02
	Total time	131s	117s	111s	107s
256^3:	Inefficiency	0.36	0.23	0.05	0.04
	Total time	281s	249s	222s	212s

The inefficiency of the three other algorithms (RSM, RSM-5, LSM) increases as the size of blocks increases. When the size of the block increases, the time to read the block from disk increases. As described previously, spending more time in I/O increases the response time to work requests, leading to higher inefficiencies. RSM is the most impacted, because it sends only one request at a time, whereas RSM-5 and LSM are sending five requests at the same time. This increases the likelihood of RSM-5 and LSM to receive work faster.

4.2.5.7 Phase 7: MPI Tasks. In this phase, we vary the number of MPI tasks, as well as the number of particles and the data size, using the following configuration:

- Data set: Fishtank
- Maximum advection steps: 10K
- Number of cells per block: 128^3
    - Test 1: 32 MPI tasks (512 cores), 5 lifelines, 1M particles, and 512 blocks.
    - Test 2: 128 MPI tasks (2048 cores), 7 lifelines 4M particles, and 2048 blocks.

∗ Test 3: 512 MPI tasks (8192 cores), 9 lifelines 16M particles, and 8192 blocks.

Table 9. Comparing the inefficiency and time of the four algorithms when varying the number MPI tasks, number of particles and number of blocks.

# MPI tasks		POP	RSM	RSM-N	LSM
32:	Inefficiency	0.20	0.14	0.03	0.02
	Total time	131s	117s	111s	107s
128:	Inefficiency	0.45	0.29	0.03	0.02
	Total time	315s	252s	209s	186s
512:	Inefficiency	0.54	0.36	0.06	0.05
	Total time	1439s	1012s	694s	671s

Table 10. Comparing the difference in workload balance between the four algorithms. The table shows the minimum work time, maximum work time, and the difference for each test. The work time in this table indicates the time spent in I/O and advection. The Diff measures the difference in time between the nodes with the highest and the lowest workloads.

# MPI tasks		POP	RSM	RSM-N	LSM
32:	Min work	82.7s	82.5s	90.1s	86.1s
	Max work	112s	101s	106s	100s
	Diff	29.3s	18.5s	15.9s	13.9s
128:	Min work	117s	112s	154s	139s
	Max work	241s	204s	186s	167s
	Diff	124s	92s	32s	28s
512:	Min work	451s	421s	525s	513s
	Max work	1209s	867s	598s	580s
	Diff	758s	446s	73s	67s

The results of this phase are presented in Table 9. The table shows that LSM reduces the inefficiency for the different test cases, with a factor of 10.8 for the largest test case.

The POP algorithm inefficiency increases as the test size increases, reaching 54% for the largest test case. As the size of the test increases, the difference in workload between the nodes increases, which can be seen in Table 10. This results in a higher load imbalance.

The POP algorithm is the most impacted by this imbalance. Using RSM reduces the inefficiency by a factor of 1.5 over POP in all cases. RSM still suffers from high inefficiencies (0.36 in the worst case). This is because the work becomes more sparse as the number of MPI task increases. Consequently, thieves are less likely to randomly find a victim with work.

RSM-5 and LSM maintain low inefficiency ratios for all cases. Using RSM-5 reduces the inefficiency over POP by 6.6 for the first test, 15 for the second test, and 9 for the third test. Using LSM reduces the inefficiency over POP by 10 for the first test, 22.5 for the second test, and 10.8 for the third test. This is because both RSM-5 and LSM can find victims with work faster by requests work from multiple victims at a time.

4.2.5.8 Summary of Findings. The evaluation showed that the LSM algorithm reduces inefficiency in all cases. The algorithm adapts the number of its lifelines (victims) as the concurrency change to make sure there is a short path from busy nodes to idle ones. Further, in our largest test case (512 ranks, 16M particles, 17B cells data sets), LSM has the lowest inefficiency of all four algorithms.

Overall, the evaluation demonstrates that the LSM algorithm is a better choice for particle advection work requesting. LSM would be particularly well suited for production visualization tools. This is because the LSM algorithm can adapt itself to better support complex cases without requiring major user inputs. Further, production visualization tools must support a large variety of use cases, including those that lead to load imbalance with traditional approaches.

4.2.6 Conclusions and Future Work.

The contribution of this section is three-fold: (1) we designed a work requesting algorithm for parallel particle advection that uses lifeline-based scheduling (LSM) method, (2) we added an extension to random scheduling (RSM-N) to use multiple victims, and (3) we evaluated the efficiency of the three scheduling methods as well as POP. As discussed in the summary of findings, our LSM algorithm improves the performance compared to traditional approaches, especially on workloads that are prone to load imbalance.

For future work, we plan to implement a multi-threaded version of the algorithm with another thread for communication. Our tests in Phase 5 and 6 show that the LSM algorithm has lower inefficiency in cases where the algorithm performed smaller work at each iteration (reading smaller blocks), which reduces the response time to a work request. We plan to test the ideas suggested by Sisneros and Pugmire [135] where the algorithm advects a portion of the particles belonging to one block, to allow the algorithm to check for work requests more frequently. We also plan to study the impact of the number of liflines (victims) on the performance. Finally, we plan to test the algorithms at larger scale.

Algorithm 2 Pseudocode for the working requesting algorithm using RSM.

```
function RSM-Advect(ParticleArray pv)
    $keepGoing \leftarrow true$
    $numActive \leftarrow totalNumberOfParticles$
    $ArrayOfParticleArrays\ \ pva[NUMBLOCKS]$
    $pva \leftarrow SortParticlesByBlock(pv)$
    $allCompletedParticles \leftarrow \emptyset$
    while keepGoing do
        $contParticles \leftarrow \emptyset$
        for i in NUMBLOCKS do
            if pva[i].size() $>$ 0 then
                $Block\ \ b \leftarrow ObtainBlock(i)$
                $ParticleArray\ \ completed,\ \ continuing$
                $(completed, continuing) \leftarrow Advect(pva[i], b)$
                $allCompletedParticles\ \ += \ \ completed$
                $contParticles\ \ += \ \ continuing$
            end if
            $MSG \leftarrow CheckForIncomingMessages()$
            if MSG = PARTICLES_TERMINATED then
                numActive -= MSG.numTerminated
            else if MSG = NEEDWORK then
                $SendWork()$
            end if
        end for
        if contParticles.size() $>$ 0 then
            $pva \leftarrow SortParticlesByBlock(contParticles)$
        else if numActive $>$ 0 & contParticles.size() = 0 then
            $randomVictim \leftarrow GetRandomVictimID()$
            $RequestWork(randomVictim)$
        else
            $keepGoing \leftarrow false$
        end if
    end while
end function
```

Algorithm 3 Pseudocode for the working requesting algorithm using Lifeline Scheduling (LSM).

```
function LSM-ADVECT(ParticleArray pv)
    $keepGoing \leftarrow true$
    $numActive \leftarrow totalNumberOfParticles$
    $ArrayOfParticleArrays\ \ pva[NUMBLOCKS]$
    $pva \leftarrow SortParticlesByBlock(pv)$
    $allCompletedParticles \leftarrow \emptyset$
    $lifelines \leftarrow CalculateLifelineGraph()$
    $numRandomReq \leftarrow 0$
    while keepGoing do
        $contParticles \leftarrow \emptyset$
        for i in NUMBLOCKS do
            if pva[i].size() $>$ 0 then
                $Block\ \ b \leftarrow ObtainBlock(i)$
                $ParticleArray\ \ completed,\ \ continuing$
                $(completed, continuing) \leftarrow Advect(pva[i], b)$
                $allCompletedParticles\ \ += \ \ completed$
                $contParticles\ \ += \ \ continuing$
            end if
            $MSG \leftarrow CheckForIncomingMessages()$
            if MSG = PARTICLES_TERMINATED then
                numActive -= MSG.numTerminated
            else if MSG = NEEDWORK then
                $SendWork()$
            end if
        end for
        if contParticles.size() $>$ 0 then
            $pva \leftarrow SortParticlesByBlock(contParticles)$
        else if numActive $>$ 0 & contParticles.size() = 0 then
            if numRandomReq $< w$ then
                $randomVictim \leftarrow GetRandomVictimID()$
                $RequestWork(randomVictim)$
                $numRandomReq++$
            else
                $RequestWork(lifelines)$
            end if
        else
            $keepGoing \leftarrow false$
        end if
    end while
end function
```

Part III

Understanding Parallel Particle Advection Behavior Over Various Workloads

This part of the dissertation is composed of two chapters. The first chapter studies the behavior of the different parallel particle advection algorithms over various workloads. The second chapter proposes an improved hybrid algorithm that is able to adapt its behavior depending on the workload.

CHAPTER V

PARALLEL PARTICLE ADVECTION BAKE-OFF

Most of the text in this chapter comes from a manuscript in preparation that is a collaboration between David Pugmire (ORNL), Abhishek Yenpure (UO), Hank Childs (UO), and myself. The implementation was mainly developed by myself with a code base that David Pugmire contributed to. Our implementation used the particle advection modules from the VTK-m software library which were developed by David Pugmire and Abhishek Yenpure. The experiments and study configurations were designed by Hank Childs and myself with help from David Pugmire.

This chapter provides a comprehensive evaluation of the most used parallel particle advection algorithms over various workloads. This work aims to understand the behavior of the different algorithms over various workloads. Our findings enable identification of the most suitable algorithm given specific workload characteristics.

5.1 Motivation

Chapter III described the different parallel particles advection solutions available in the literature. These solutions are still actively investigated and several optimizations have been proposed by the visualization community. Each of these solutions behaves differently depending on the characteristics of the workload, which can lead to poor performance in some cases. Therefore it is critical for users to select the appropriate algorithm for their particular workloads. In this chapter, we compare four of the most used parallel particle advection algorithms over various workloads and determine the most suitable algorithms for each workload.

5.2 Experiment Overview

This section describes the details of our study.

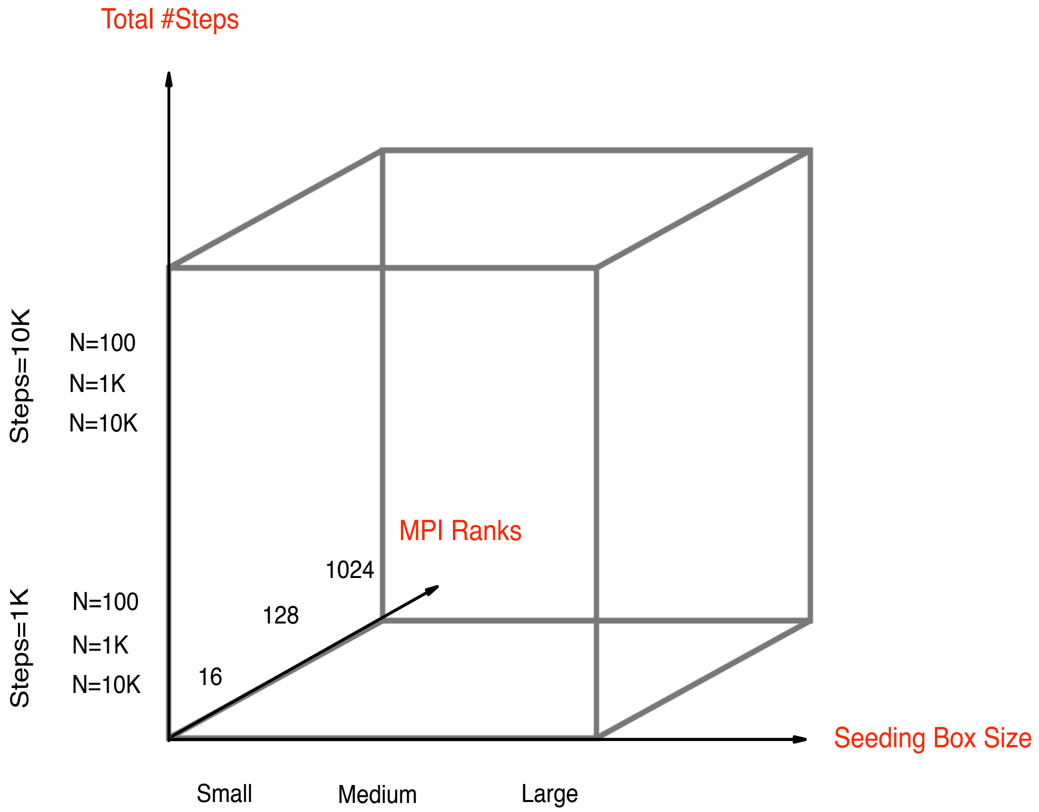


Figure 15. Three axes for defining a workload: total number of steps, size of seeding box, and number of MPI ranks.

5.2.1 Algorithm Comparison Factors.

There are three main axes to our study, presented in Figure 15:

- Total number of steps (6 options)
- Size of seeding box (3 options)
- Number of MPI ranks (3 options)

In total,we considered 54 (=6*3*3) configurations. We tested each configuration with all four algorithms, meaning 216 (=54*4) experiments overall. The following subsections discuss the impact of each one of these axes as well as their configurations.

5.2.1.1 Total Number of Steps. The total number of steps represents the amount of work defined as the product of the number of particles and the maximum number of advection steps. Different flow visualization algorithms require different representations. Some algorithms require a small number of particles that advect for a long duration, while others require a large number of particles that advect for a short duration.

We consider three options for the number of particles and represent it in the form pf 1 particle for each C cells. We consider seed a particle for every 100 cells, every 1000 cells, and every $10,000$ cells, and denote these as $P/100C$, $P/1KC$, and $P/10KC$, respectively ("$P/1KC$" meaning 1 particle for every one thousand cells). As the value of C decreases, the density of particles per cell increases, therefore increasing the total number of particles. We consider 2 options for the duration of particles (maximum advection steps), which are 1K and 10K.

Consider an example where the data set size is $1024 * 1024 * 512$. If we seed according to $P/10KC$, then we would have a particle for every $10,000$ cells. Since the total number of cells is approximately $537M$, then the number of particles will be approximately $54K$. Further, if the duration is 1000 steps, then the total number of advection steps would be $54M$ ($54K$ particles $\times$ 1000 steps per particle).

5.2.1.2 Size of Seeding Box. The size of the seeding box represents the particle distribution. This impacts the I/O cost and can impact load balance. If the distribution is dense i.e., all the seeds originate in a small box, only a subset of the data set will be required, which reduces the cost of I/O. Figure 16 shows the three different boxes considered in this study.

5.2.1.3 Number of MPI Tasks. We test the weak scalability of the algorithms by varying the number of MPI tasks, as well as the number of data

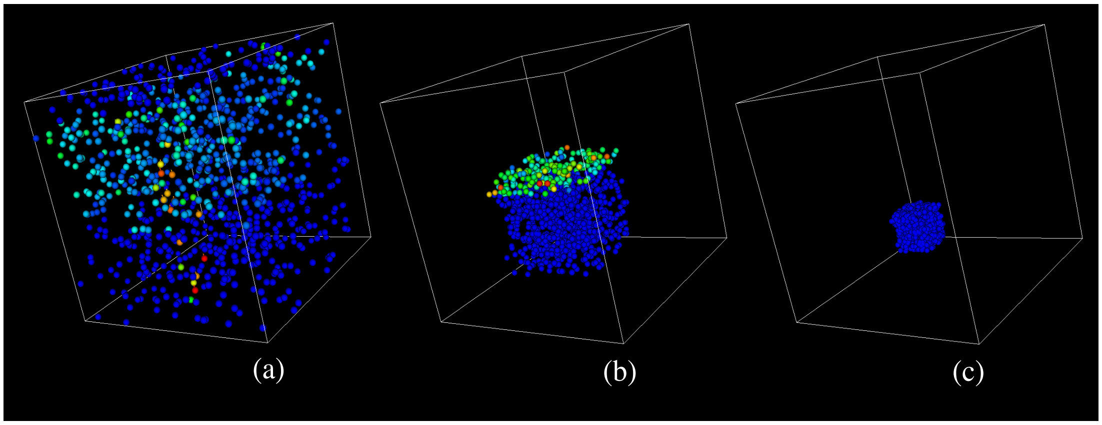


Figure 16. The three seeding boxes considered in the study: (a) large box, (b) medium box, and (c) small box.

blocks. The size of each data block is 128^3, and three levels of concurrency are considered:

- **Concurrency1:** 16 MPI tasks, 4 tasks per node, 8 cores per task (128 cores), and 256 blocks.
- **Concurrency2:** 128 MPI tasks, 4 tasks per node, 8 cores per task (1024 cores), and 2,048 blocks.
- **Concurrency3:** 1024 MPI tasks, 4 tasks per node, 8 cores per task (8192 cores), and 16,384 blocks.

5.2.2 Data Set. We use the “Fishtank” data set for our study, which comes from a thermal hydraulics simulation by the NEK5000 [132] code. In this particular simulation, twin inlets pump water of differing temperatures into a box, and the vector field captures the fluid flow within the box. The simulation’s focus is on understanding mixing behavior, as well as temperature at the box’s outlet.

5.2.3 Algorithm Setting. Many of the algorithms have “knobs” to optimizing their performance. In our study, we used the following values:

- **Cache size:** The cache size in our study is 25 blocks per node, where each block has approximately two million cells (for details see Section 4.1.)
- **Number of lifelines:** The number of lifelines for the LSM algorithm is computed as the following $log_2(\#Ranks)$ (for details see Section 4.2.)
- **Group size:** The group size for the MW algorithm varies depending on the number of MPI tasks (for details see Section 3.2.4). We tried different group sizes and found out that the best results are when there are 4 masters:
  - 16 Ranks: group size is 4, which means there are 4 masters.
  - 128 Ranks: group size is 32, which means there are 4 masters.
  - 1024 Ranks: group size is 256, which means there are 4 masters.

5.2.4 Hardware Used. The study was run on Cori at Lawrence Berkeley National Laboratory's NERSC facility. It contains 2,388 Intel Xeon "Haswell" processor nodes. Each node has two 2.3 GHz 16-core processors, each core supports 2 hyper-threads and there is 128 GB of memory per node.

5.2.5 Performance Measurement. We measure the performance of the different algorithms by calculating the number of steps computed per rank per second. The higher the number of steps is, the more efficient the algorithm is, since it indicates lower execution time. Our metric works as follows. Let S_T be the total number of advection steps for the workload, T be the total execution time for the slowest rank, and N be the number of ranks. Then we define the number of steps per rank per second with the following equation: We define the number of steps per rank per second with the following equation:

$$\text{Number of steps per rank per second} = \frac{\mathrm{S_T}}{(\mathrm{T} * \mathrm{N})}$$

We will refer to this measurement as $SPRPS$ in our study. We also provide the corresponding execution time for each experiment in addition to the $SPRPS$.

5.3 Testing Infrastructure

This section describes the testing infrastructure of our study. It is organized as follows: foundational algorithmic concepts (5.3.1), carrying out advection work (5.3.2), and communication between nodes (5.3.3).

5.3.1 Foundational Algorithmic Concepts. All four algorithms described share common elements. First, they start by generating the seeds and distributing them among compute nodes. Then, in all four algorithms, each node executes the main loop which is composed of a worker function and a communication function. The worker function performs the I/O operations, the advection, and the processing of particles after each advection round. The communication function sends and receives data. This data can be particles or messages. The algorithm completes when all particle trajectories are calculated. Pseudocode 4 describes the general program that runs identically on each node for all four algorithms.

The pseudocode uses the following building blocks:

- GenerateSeeds(): a function that generates the initial seeds.
- WorkerFunction(): a function that performs I/O operations, advection and process the particles after advection.
- CommunicationFunction(): a function that sends and receives data (particles or messages) between nodes.

Algorithm 4 Pseudocode for the general skeleton of the four algorithms.

```
$numActive \leftarrow TotalNumParticles$
$activeParticles \leftarrow GenerateParticles()$
while $numActive > 0$ do
    if $activeParticles.size() > 0$ then
        $WorkerFunction(activeParticles)$
    end if
    $CommunicationFunction(activeParticles)$
end while
```

The implementation for these functions varies depending on the algorithm.

5.3.2 Worker Function. The worker function is responsible for executing three operations: 1) I/O, 2) advection, and 3) processing particles.

The I/O operation varies depending on the algorithm; it can be either a static allocation or load on demand. If the algorithm uses static allocation, then each node only reads the blocks assigned to it. If the algorithm uses load on demand, then each node loads the data blocks as needed. The POD algorithm uses static allocation, while the POP and LSM algorithms use load on demand. In the MW algorithm, at each iteration workers load data depending the rules instructed by the master, which can be a static allocation or load on demand.

In all four algorithms, each node passes the particles in the current data block to the VTK-m [2, 3] routine. The VTK-m routine will perform the advection either in serial, or using on node parallelism, depending on the setting. For our implementation we use on node parallelism using the Intel Threading Building Blocks [136].

Finally, each node processes the particles after advection, and this process has similarities and some variations for different algorithms. For all four algorithms, each node terminates the particles that reached the maximum number of advection steps or exited the data set. Each node also notifies the other nodes of the number of terminated particles. The four algorithms vary in the way they handle particles that exited the current data block. In the POD algorithm, the node stores the particle in a *communicate* queue to be sent to other nodes. In the POP and LSM algorithms, the node stores the particle in an *inActive*, which will be processed after advecting all the particles from the current block. In the MW algorithm, at each iteration, workers either communicate particles or store the particles in an *inActive* queue and load the needed data block.

5.3.3 Communication Function.

The communication function is responsible for sending and receiving data, which can be particles or messages.

We built a communication routine that uses the Message Passing Interface [137] for communication across the nodes. The routine uses a non-blocking communication, and can communicate messages and particles. It takes care of serializing and de-serializing the data.

Different algorithms communicate different types of data. In the four algorithms, each node sends a $TERMINATE$ message to other nodes to notify them with the number of particles it terminated. In the POD algorithm, nodes communicate particles according to the data block assignment. The POP algorithm does not exchange any other data except for the $TERMINATE$ message. In the LSM algorithm, when a node finishes its workload it sends a $NEED\_PARTICLE$ message requesting work from a victim node. In addition, nodes communicate particles when a thief node steals work from a victim node. In the MW algorithm, nodes exchange different types of messages. The workers can request work or a needed data block from the master by sending the messages $NEED\_PARTICLE$ or $NEED\_BLOCK$, respectively. At the end of each iteration, the worker updates the master with information about its status, including the block IDs currently loaded and the number of particles it has. This information are used by the master when making work assignment. The master can send two types of messages to the worker: $SEND\_PARTICLES$, and $LOAD\_BLOCK$. The $SEND\_PARTICLES$ message instructs a worker to send particles to another worker, while the $LOAD\_BLOCK$ message instructs the worker to load a new block.

5.4 Results

This section presents the results of our study. The section is broken into 7 sub-sections. Sub-section 5.4.1 discusses how to interpret figures, and section 5.4.2 discusses common behaviors across all algorithms. Sub-sections 5.4.3 through 5.4.6 discuss individual algorithms. Finally, sub-section 5.4.7 performs a comparative analysis, contrasting the performance behaviors between the algorithms.

5.4.1 Figure Representation. Figures 17, 19, 21, and 23 present per-algorithm results, each using the same format. For each Figure, the X axis represents Particle for every C Cells * #Steps per Particle. For example, $P/10KC * 1K$ means there is one particle for every 10K cells and the duration for each particle is 1K advection steps. The leftmost value is the smallest number of total advection steps and the rightmost is the largest. However, some workloads are equal: 1) $P/10KC * 10K$ is equal to $P/1KC * 1K$ and 2) $P/1KC * 10K$ is equal to $P/100C * 1K$. The Y axis represents our performance metric: the number of steps per rank per second (see Section 5.2.5).

Figures 18, 20, 22, and 24 present per-algorithm results, each using the same format. These figures have the same X axis as the previous figures. The Y axis represents the execution time in seconds.

5.4.2 Common Behaviors.

The results in Figures 17, 19, 21, and 23 show some common behaviors shared by the four algorithms, which are the following:

- For two tests that have the same number of particles (for example test1: $P/10KC * 1K$ and test2: $P/10KC * 10K$), the performance is better when the duration is longer. This is because the total work done (advection step) per particle is increased, which offsets the cost of paying the operations to handle more particles.
- For two tests that have the same amount of work (total number of advection steps) the performance is better when the number of particles is smaller and the duration is larger. For example, test1: $P/10KC * 10K$, and test2: $P/1KC * 1K$ have the same amount of work. Yet, the performance of test1 is better than test2 and this is again due to the extra cost of adding more particles.
- For all different workloads, the performance decreases as the scale increases. Note that if these algorithms achieved weak scalability, then we would expect performance to be constant. As the number of rank increases, the number of data blocks (size of the data) and the number of particles increase (see Section 5.2.1.3). Increasing the number of blocks often increases the need for I/O or communication operations as particles are more likely to move from a block to the next.

5.4.3 POD Behavior.

The results presented in Figure 17 show the performance of POD in $SPRPS$ for the different workloads, as well as the scalability of the algorithm. The algorithm performs well when the seeding box is

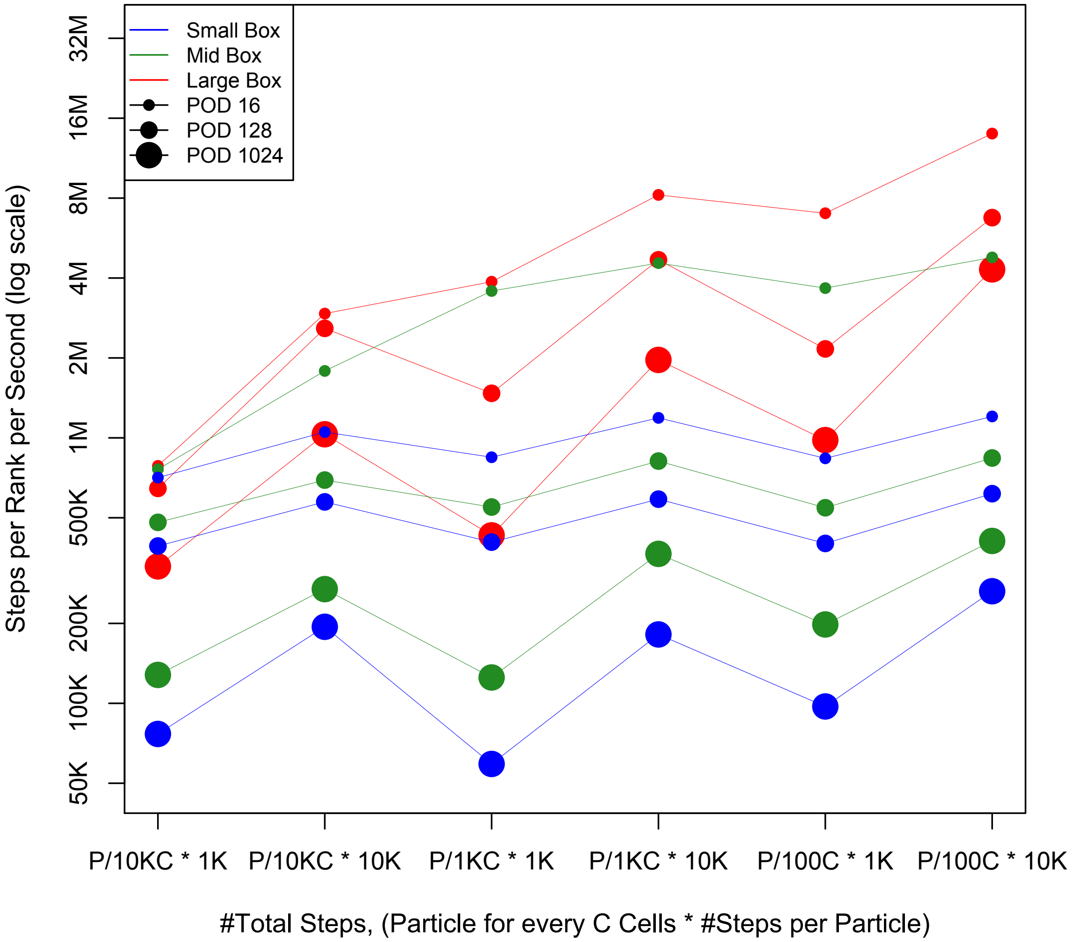


Figure 17. The performance scalability of the parallelize over data algorithm for different workloads.

large. It obtains the best performance using 16 ranks, $P/100C * 10K$ particles, where the algorithm reaches a performance of 14 million $SPRPS$ for a large seeding box workload. The performance drops as the size of the seeding box gets smaller, where it is computing 4 million $SPRPS$ for the middle box case, and 1 million $SPRPS$ for the small box case. This is a reduction of 14X between the best case and worst case. The decrease in the performance is due to the load

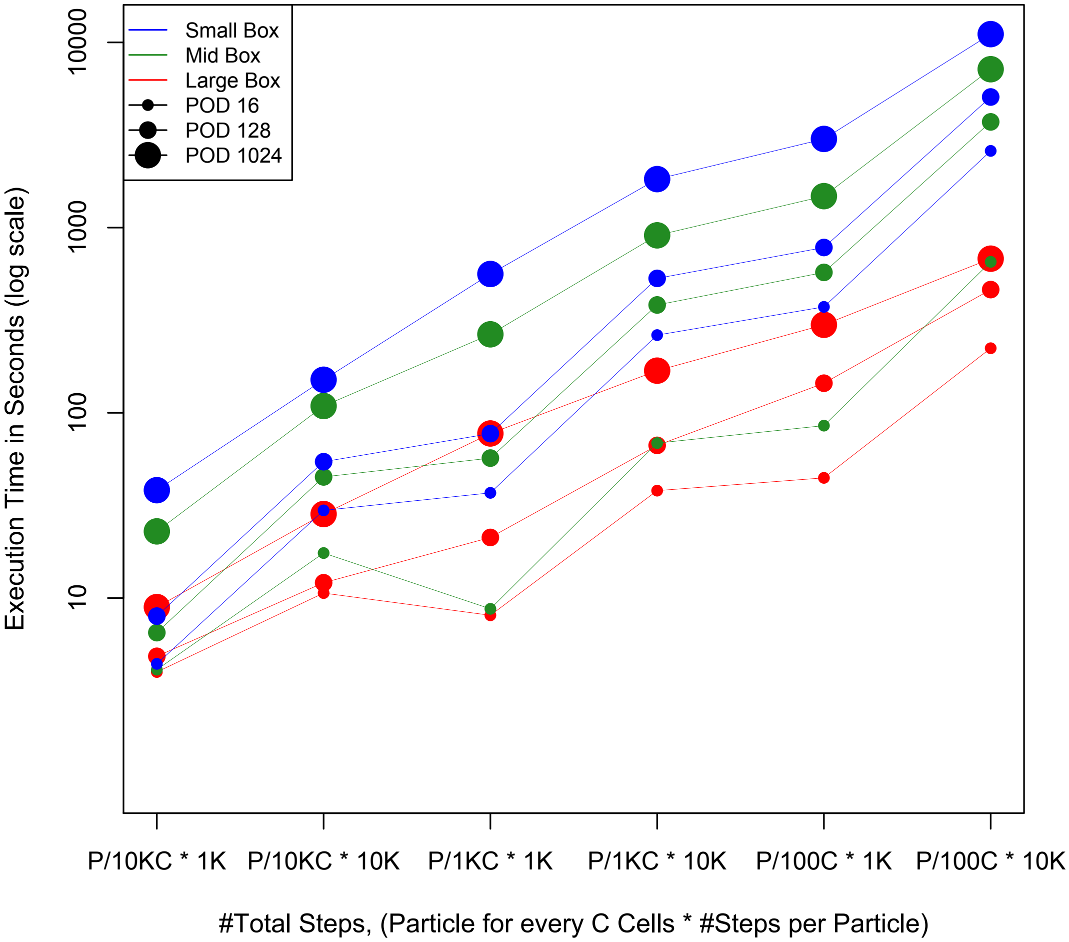


Figure 18. The performance scalability of the parallelize over data algorithm for different workloads.

imbalance in workloads between different ranks. This is because ranks that are responsible for the data blocks inside the seeding box will be doing the work, while other ranks will remain idle during the entire execution time. This pattern becomes more significant at a larger scale, since the load imbalance becomes more significant, where the performance drops down to 600 thousand $SPRPS$ for the second concurrency (128 ranks) and down to 200 thousand $SPRPS$ for the largest concurrency (1024 ranks).

The results presented in Figure 18 show the execution time of POD for the different workloads, as well as the scalability of the algorithm. These results correspond to the execution time of the experiments presented in Figure 17. The algorithm has low execution time for large seeding box workloads, but as the size of the box gets smaller, the execution time increases. The execution time also increases as the scale increases. Looking at the best workload for POD, which is the workload with the largest number of steps and large seeding box workload. The time increases from 223 seconds to 462 seconds when scaling from the first concurrency (16 ranks) to the second concurrency (128 ranks), which is an increase of 2.1X. When scaling from the second concurrency (128 ranks) to the largest concurrency (1024 ranks), the execution time increases from 462 to 679 seconds, which is an increase of 1.5X. The scalability of the algorithm gets worse for the small seeding box workload. The time increases from 2595 seconds to 5071 seconds when scaling from the first concurrency (16 ranks) to the second concurrency (128 ranks), which is an increase of 2X. When scaling from the second concurrency (128 ranks) to the largest concurrency (1024 ranks), the execution time increases from 5071 to 11082 seconds, which is an increase of 2.2X.

5.4.4 POP Behavior.

The results presented in Figure 19 show the performance of POP in $SPRPS$ for the different workloads, as well as the scalability of the algorithm. Unlike POD, the POP algorithm performs well for small seeding size, due to the reduction in I/O cost. It obtains the best performance using 16 ranks, $P/100C * 10K$ particles, where the algorithm reaches a performance of 14 million $SPRPS$ for a small seeding box workload. As the

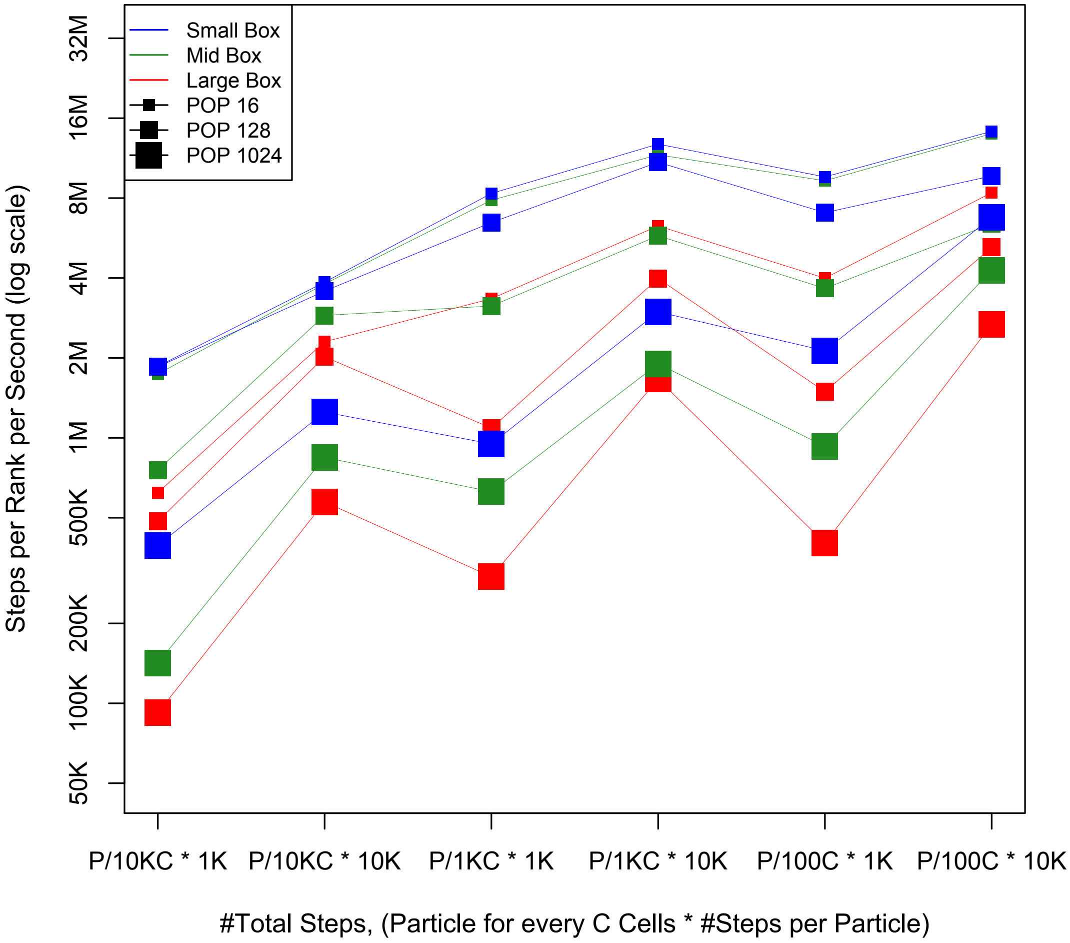


Figure 19. The performance scalability of the parallelize over particles algorithm for different workloads.

size of the box increases, the performance drops because of the increase in I/O operations, dropping down to 8 million *SPRPS* for the large box workload at its worst case. This is a reduction of 1.75X between the best case and worst case. This pattern becomes more significant at a larger scale, since the size of the data and the number of particles increase. The performance drops to 5 million *SPRPS* for

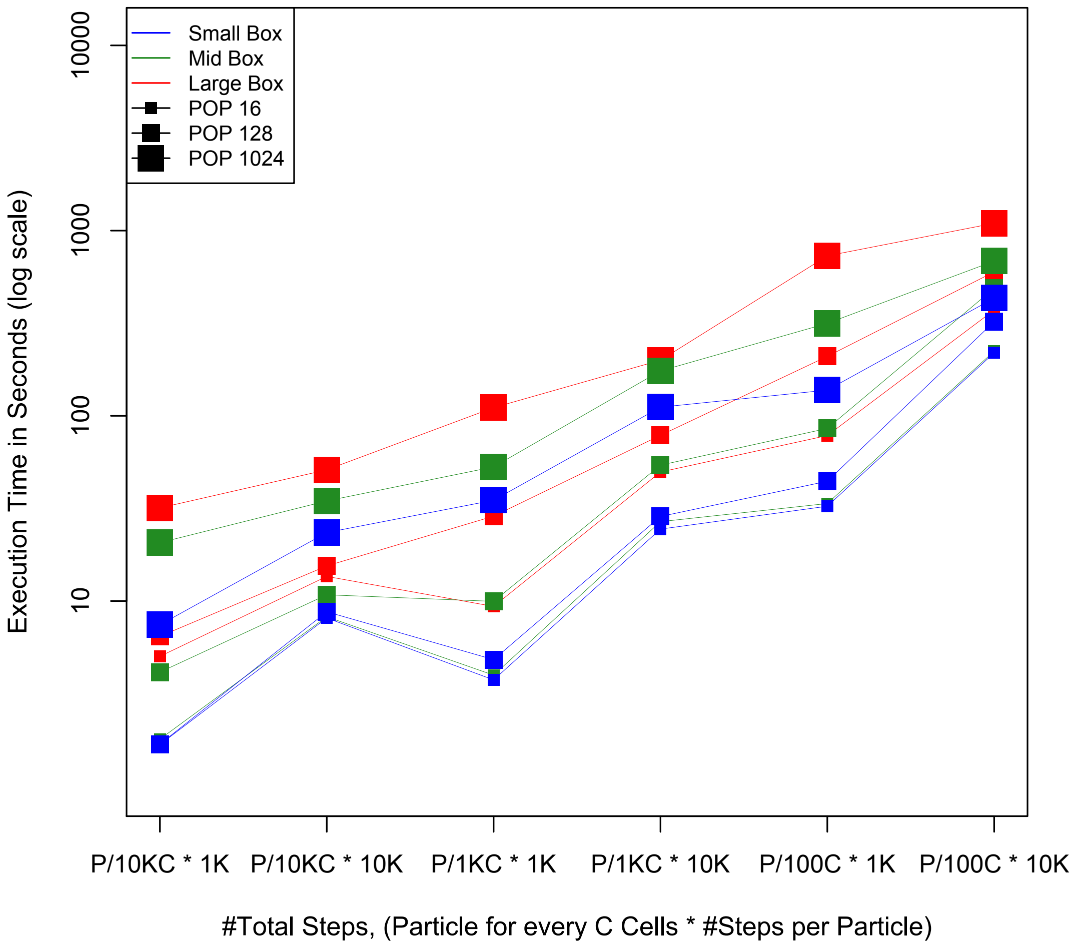


Figure 20. The performance scalability of the parallelize over particles algorithm for different workloads.

the second concurrency (128 ranks) and down to 2 million $SPRPS$ for the largest concurrency (1024 ranks).

The results presented in Figure 20 show the execution time of POP for the different workloads, as well as the scalability of the algorithm. These results correspond to the execution time of the experiments presented in Figure 19. The algorithm has low execution time for small seeding box workloads, but as the size of the box gets larger, the execution time increases. The execution time also increases as the scale increases. Looking at the best workload for POP, which is the workload with the largest number of steps and small seeding box workload. The time increases from 200 seconds to 322 seconds when scaling from the first concurrency (16 ranks) to the second concurrency (128 ranks), which is an increase of 1.16X. When scaling from the second concurrency (128 ranks) to the largest concurrency (1024 ranks), the execution time increases from 322 to 432 seconds, which is an increase of 1.5X. The scalability of the algorithm gets worse for the large seeding box workload. The time increases from 372 seconds to 598 seconds when scaling from the first concurrency (16 ranks) to the second concurrency (128 ranks), which is an increase of 1.6X. When scaling from the second concurrency (128 ranks) to the largest concurrency (1024 ranks), the execution time increases from 598 to 432 seconds, which is an increase of 1.83X.

5.4.5 LSM Behavior. The results presented in Figure 21 show the performance of LSM in $SPRPS$ for the different workloads, as well as the scalability of the algorithm. Similar to POP, the LSM algorithm performs well for small seeding box size, due the reduction in I/O cost. It obtains the best performance using 16 ranks, $P/100C * 10K$ particles, where the algorithm reaches a performance of 16 million $SPRPS$ for a small seeding box workload. As the size of the box increases, the performance drops because of the increase in I/O operations, dropping down to 10 million $SPRPS$ for the large box workload at its worst case. This is a reduction of 1.6X between the best case and worst case. This pattern becomes more significant at a larger scale, since the size of the data and the number of particles increase. The performance drops to 6 million $SPRPS$ for

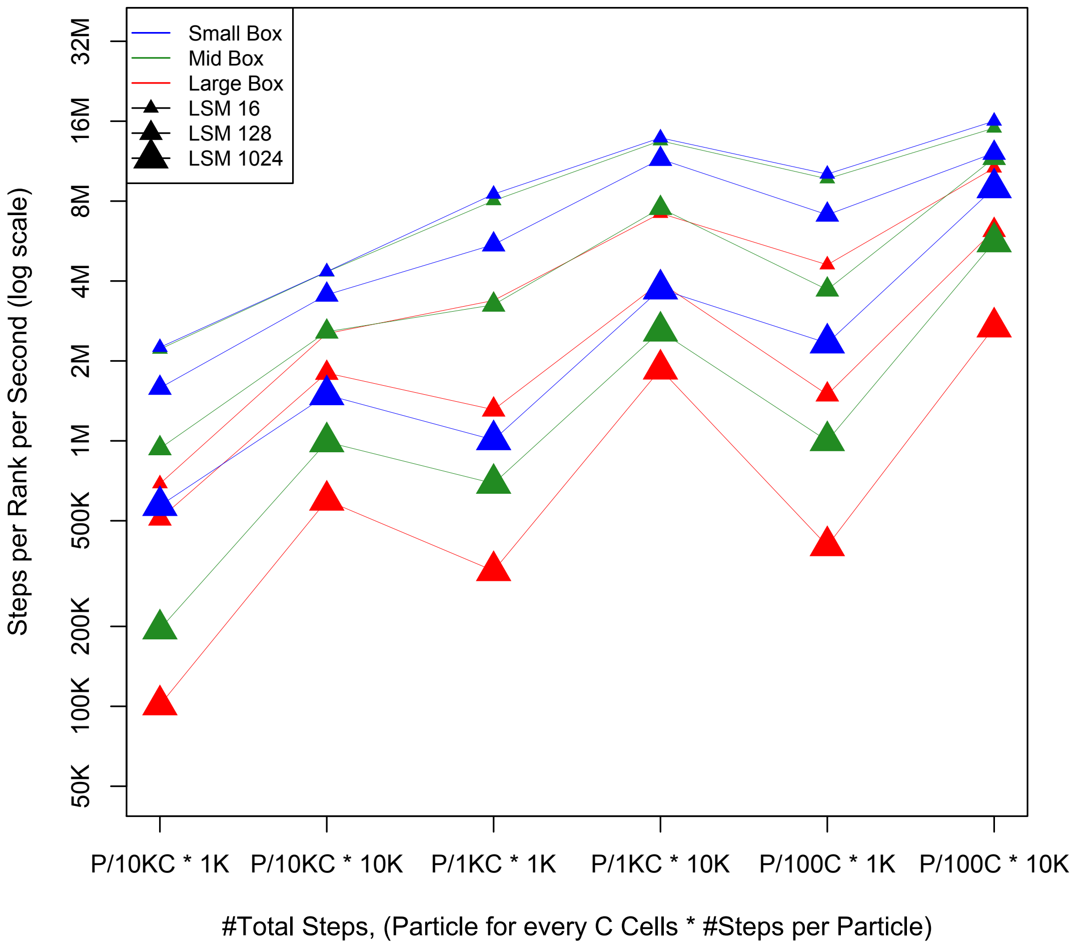


Figure 21. The performance scalability of the work requesting algorithm for different workloads.

the second concurrency (128 ranks) and down to 2 million $SPRPS$ for the largest concurrency (1024 ranks).

The results presented in Figure 22 show the execution time of LSM for the different workloads, as well as the scalability of the algorithm. These results correspond to the execution time of the experiments presented in Figure 21. The algorithm has low execution time for small seeding box workloads, but as the

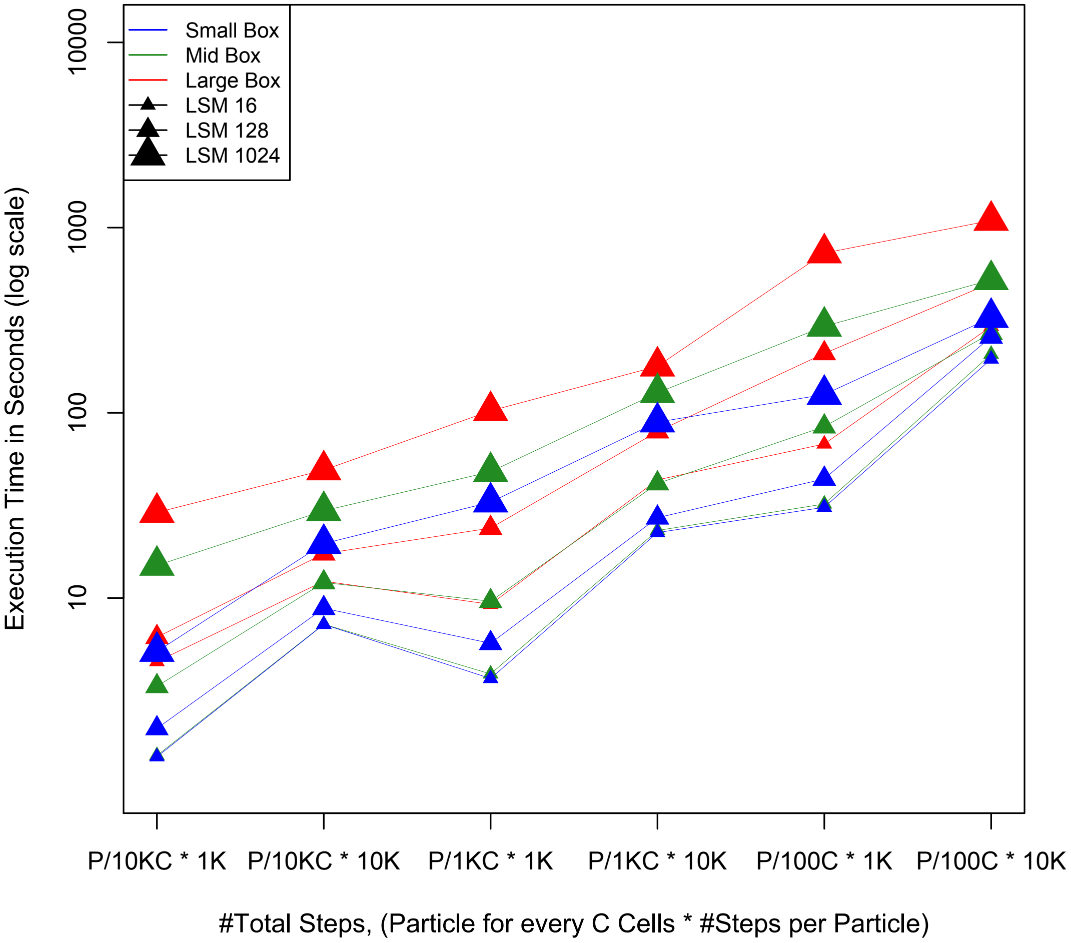


Figure 22. The performance scalability of the work requesting algorithm for different workloads.

size of the box gets larger, the execution time increases. The execution time also increases as the scale increases. Looking at the best workload for LSM, which is the workload with the largest number of steps and small seeding box workload. The time increases from 195 seconds to 257 seconds when scaling from the first concurrency (16 ranks) to the second concurrency (128 ranks), which is an increase of 1.3X. When scaling from the second concurrency (128 ranks) to the largest concurrency (1024 ranks), the execution time increases from 257 to 326 seconds, which is an increase of 1.26X. The scalability of the algorithm gets worse for the large seeding box workload. The time increases from 293 seconds to 502 seconds when scaling from the first concurrency (16 ranks) to the second concurrency (128 ranks), which is an increase of 1.7X. When scaling from the second concurrency (128 ranks) to the largest concurrency (1024 ranks), the execution time increases from 502 to 432 seconds, which is an increase of 2.2X.

The LSM algorithm performs better than POP because of its ability to balance the workload better by requesting work from other ranks (for more details, see our study in Section 4.2). Further, its overhead to locate victims does not affect overall performance.

5.4.6 MW Behavior.

The results presented in Figure 23 show the performance of MW in $SPRPS$ for the different workloads, as well as the scalability of the algorithm. The algorithm performs better for smaller box seeding due to the decrease of I/O cost. The algorithm obtains the best performance using 16 ranks, $P/100C * 10K$ particles, where the algorithm reaches a performance of 8 million $SPRPS$ for a small seeding box workload. As the size of the box increases, the performance slightly drops because of the increase in I/O operations, dropping down to 7 million $SPRPS$ for the large box workload at its worst case. This is a reduction of 1.1X between the best case and worst case. This pattern becomes more significant at a larger scale, since the size of the data, the number of particles, and number of MPI ranks increases. The performance drops to 5 million $SPRPS$ for the second concurrency (128 ranks) and down to 1 million $SPRPS$ for the largest concurrency (1024 ranks).

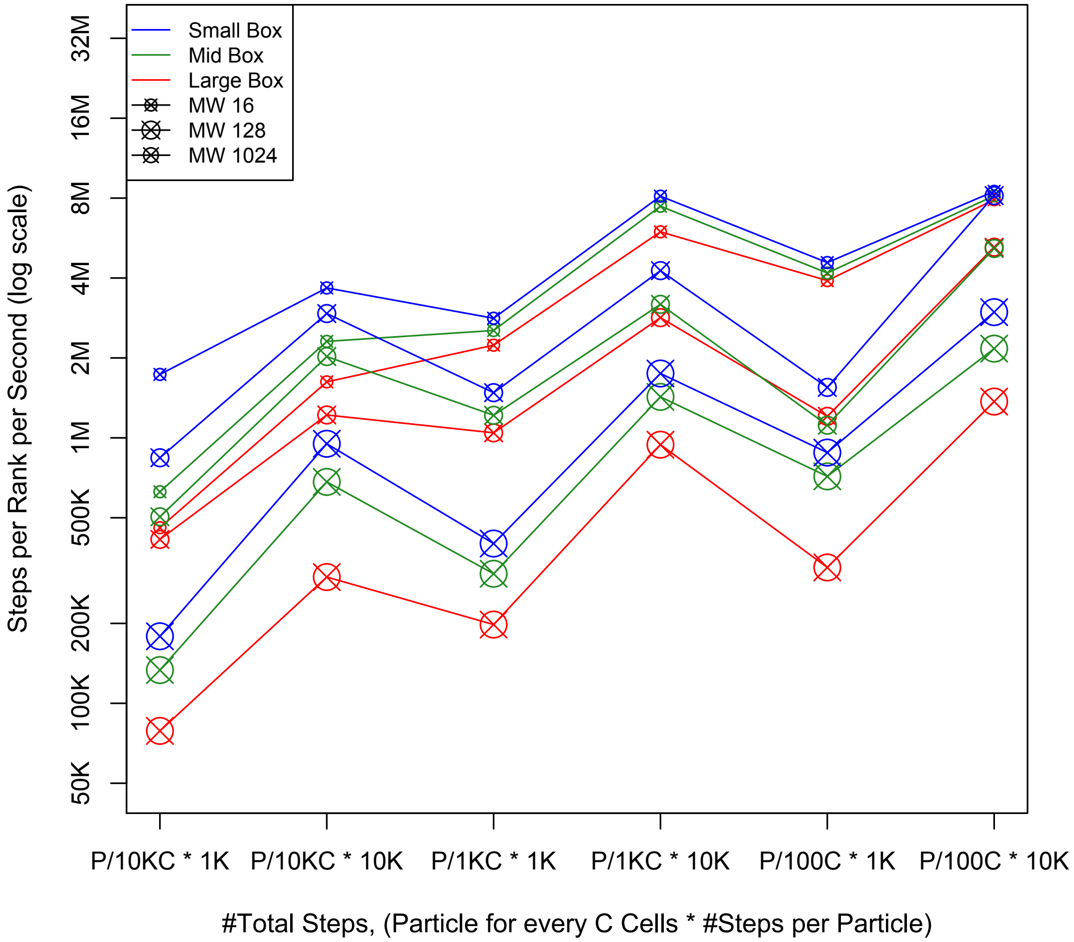


Figure 23. The performance scalability of the master/worker algorithm for different workloads.

The results presented in Figure 24 show the execution time of MW for the different workloads, as well as the scalability of the algorithm. These results correspond to the execution time of the experiments presented in Figure 23. The algorithm has low execution time for small seeding box workloads, but as the size of the box gets larger, the execution time increases. The execution time also increases as the scale increases. Looking at the best workload for MW, which is

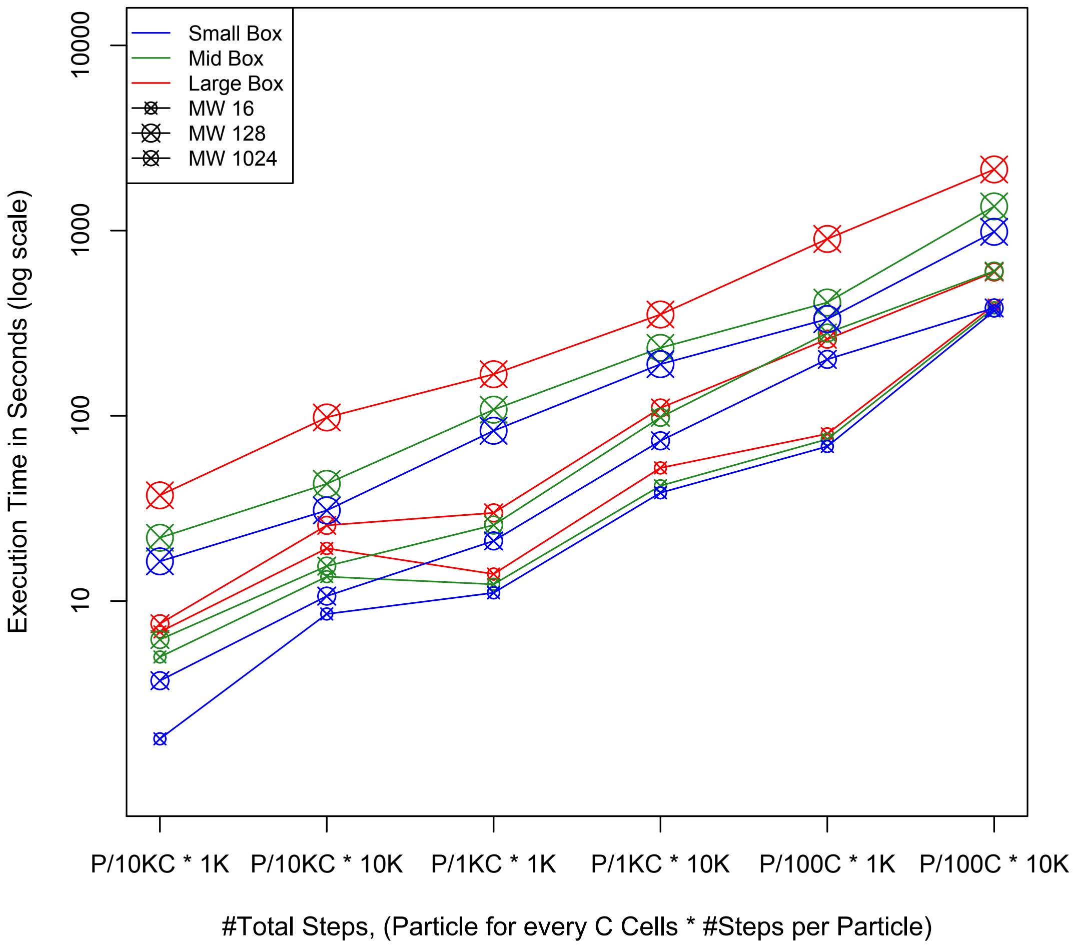


Figure 24. The performance scalability of the master/worker algorithm for different workloads.

the workload with the largest number of steps and small seeding box workload. The time increases from 396 seconds to 597 seconds when scaling from the first concurrency (16 ranks) to the second concurrency (128 ranks), which is an increase of 1.5X. When scaling from the second concurrency (128 ranks) to the largest concurrency (1024 ranks), the execution time increases from 597 to 2138 seconds, which is an increase of 3.6X. The scalability of the algorithm gets worse for the large seeding box workload. The time increases from 368 seconds to 382 seconds when scaling from the first concurrency (16 ranks) to the second concurrency (128 ranks), which is an increase of 1.03X. When scaling from the second concurrency (128 ranks) to the largest concurrency (1024 ranks), the execution time increases from 382 to 984 seconds, which is an increase of 2.6X.

The algorithm performs worse than the other three algorithms due to 1) the idle time workers spend waiting for the master and 2) the extra communication time between workers and their masters.

5.4.7 Comparative Analysis. For all four algorithms, the best performance was achieved at the 16 Ranks, $P/100C * 10K$. Table 11 shows the different seeding box sizes and the performance for the best case for each of the four algorithms. The LSM algorithm achieves the best performance overall between all the algorithms with the case of small seeding box with 16 ranks. This is because the algorithm is designed to balance the workload between different nodes. Both the POD and the POP algorithm have similar performances for their best case. Finally, the MW algorithm has the worst performance from all four algorithms. This is caused by the additional communication cost between masters and workers and the idle time that workers spent waiting for the master.

Table 11. This table shows the best case scenario for each of the four algorithms, i.e., what seeding box size yielded the highest $SPRPS$ for an algorithm, what was that $SPRPS$, and their execution time.

	POD	POP	LSM	MW
Seeding box size	Large	Small	Small	Small
Performance in $SPRPS$	14M	14M	16M	8M
Execution time in seconds	223	218	195	368

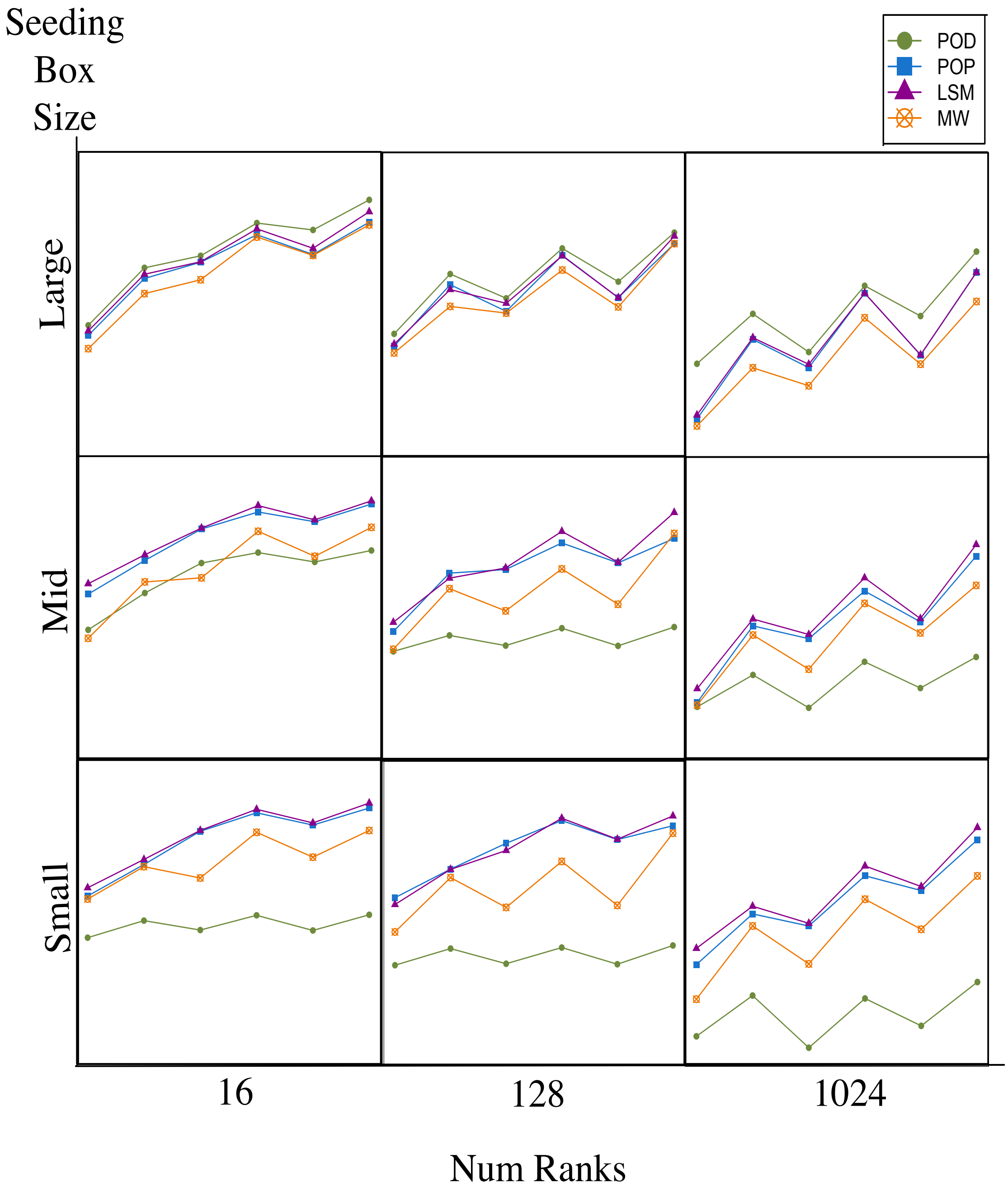


Figure 25. Comparing the performance of the four algorithms for different workloads.

An important goal of this dissertation is to determine the best algorithm for each workload. Table 12 shows the best algorithm for the different seeding box sizes and the performance of that algorithm at different scales. The LSM algorithm has the best performance for the small and medium box workloads. For a medium box workload the performance drops by 1.3X when scaling from 16 to 128 ranks and drops by 2.2X when scaling from 128 to 1024 ranks. For a small box workload the performance drops by 1.3X when scaling from 16 to 128 ranks and drops by 1.5X when scaling from 128 to 1024 ranks. The POD algorithm has the best performance for the large box workload. The performance of the algorithm drops by 2.4X when scaling from 16 to 128 ranks and drops by 1.5X when scaling from 128 to 1024 ranks. This drop is caused by the extra communication cost.

Figure 25 shows the performance of the four algorithms for different workloads. There are two main observations that can be seen from the figure. The first one is how these algorithms compare to each other. The results show that, for the large seeding box, the performance of the algorithm from best to worst is as follows: POD, LSM and POP, and then MW. The performance for medium seeding box from best to worst is as follows: LSM, POP, MW, and then POD (for most cases). Finally the performance for the small seeding box from best to worst is as follows: LSM, POP, MW, and then POD. The second observation is the reduced performance for these algorithms at larger scale. The drop down at large scale is either because of additional I/O operations, additional communication, or increase in load imbalance. In the case of the large box, each algorithm is impacted negatively due to additional I/O operations, or extra communication. However, the impact on their performance remains limited because the workload tends to be evenly distributed with a large seeding box. On the other hand, with a medium and small seeding box, the performance of the POD degrades considerably with scale due to the increase of load imbalance. This shows that the negative impact of load imbalance on the performance is larger than the impact of additional I/O. All the other algorithms except POD are better equipped to load balance the workload at larger scale and therefor suffer less as the scale increase.

Table 12. The best algorithm for the different seeding box sizes, the table shows the best suitable algorithm and the performance of that algorithm as different scales.

		Small Box	Mid Box	Large Box
Best algorithm		LSM	LSM	POD
Performance in *SPRPS*:	16 Ranks	16M	15M	14M
	128 ranks	12M	11M	6M
	1024 ranks	8M	5M	4M

5.5 Summary of Findings

This section summarizes the findings from our study:

- Using POD for small box workloads leads to bad performance due to load imbalance. As the size of the seeding box gets smaller the performance decreases because of the increase in load imbalance.
- Using POP and LSM for small box workloads reduces I/O, which increases the efficiency and thus reduces the execution time. As the size of the seeding box gets smaller the performance increases because the number of I/O operations decreases.
- The POP and LSM algorithms have similar performances.
- The smaller the box is the better the performance of MW. This is because there is less I/O operations and the algorithm distributes the workload between the different workers.
- The performance of MW is lower than the other algorithms because of the additional communication time and the idle time caused by workers waiting for the master.

- The cost of load imbalance is larger than the cost of I/O.
- All algorithms performs poorly for the smallest workload since there is not enough work to offset the cost of I/O and communication.

CHAPTER VI

HYLIPOD: IMPROVED HYBRID PARALLEL PARTICLE ADVECTION ALGORITHM

6.1 Motivation

The results of the bake-off study in Chapter V showed that the two algorithms that had the best performance are POD and LSM. However, each one of them had some workloads where they performed poorly. The MW algorithm has the advantage of being hybrid so it can adapt its behavior. But its main disadvantage is the idle time cause by additional communication between masters and workers. We propose a hybrid algorithm between the POD and LSM, which we call HyLiPod. The algorithm adapts its behavior depending on the workload and chooses between POD and LSM to get the best possible performance. Unlike the MW algorithm, our algorithm does not have a master/workers structure so it avoids the extra communication and idle time that MW suffers from.

6.2 Algorithm

In the beginning of the program, the algorithm calls a function that takes the seeding box and returns which blocks are included. This information is used to run either as POD or LSM. Pseudocode 5 describes our algorithm.

The pseudocode uses the following building blocks:

- GetBlocksInBox(): a function that returns a list of block ids that are within the seeding box boundaries.
- GenerateSeeds(): a function that generates the initial seeds.
- WorkerFunction(): a function that performs I/O operations, advection and process the particles after advection.

Algorithm 5 Pseudocode for the HyLiPod algorithm.

```
$ListBlockIdInBox \leftarrow GetBlocksInBox(seedingBox)$
$blocksIncludedPercent \leftarrow ListBlockIdInBox.size()/totalNumBlocks$
if $blocksIncludedPercent >= 0.7$ then
    $algo \leftarrow POD$
else
    $algo \leftarrow LSM$
end if
$numActive \leftarrow TotalNumParticle$
$activeParticles \leftarrow GenerateParticles(ListBlockIdInBox)$
while $numActive > 0$ do
    if $activeParticles.size() > 0$ then
        $WrokerFunction(activeParticles, algo)$
    end if
    $CommunicationFunction(activeParticles, algo)$
end while
```

- CommunicationFunction(): a function that sends and receives data (particles or messages) between nodes.

6.3 Experiments Overview

We used the same configurations as Chapter V but we the limited scalability to 128 ranks as the algorithm performance is the same as either POD or LSM.

We computed the percentage of of blocks within the seeding box boundaries to the total number of blocks. That number was used to determine the algorithm behavior by comparing it with a given threshold. In this study, we use a threshold we of 70%.

6.4 Results

This section discusses the results of our study.

6.4.1 HyLiPoD Behavior. The results presented in Figure 26 show the performance of HyLiPoD for the different workloads, as well as the scalability of the algorithm. The figure is represented as described in Section 5.4.1. The algorithm performs better for small box since there are less blocks included,

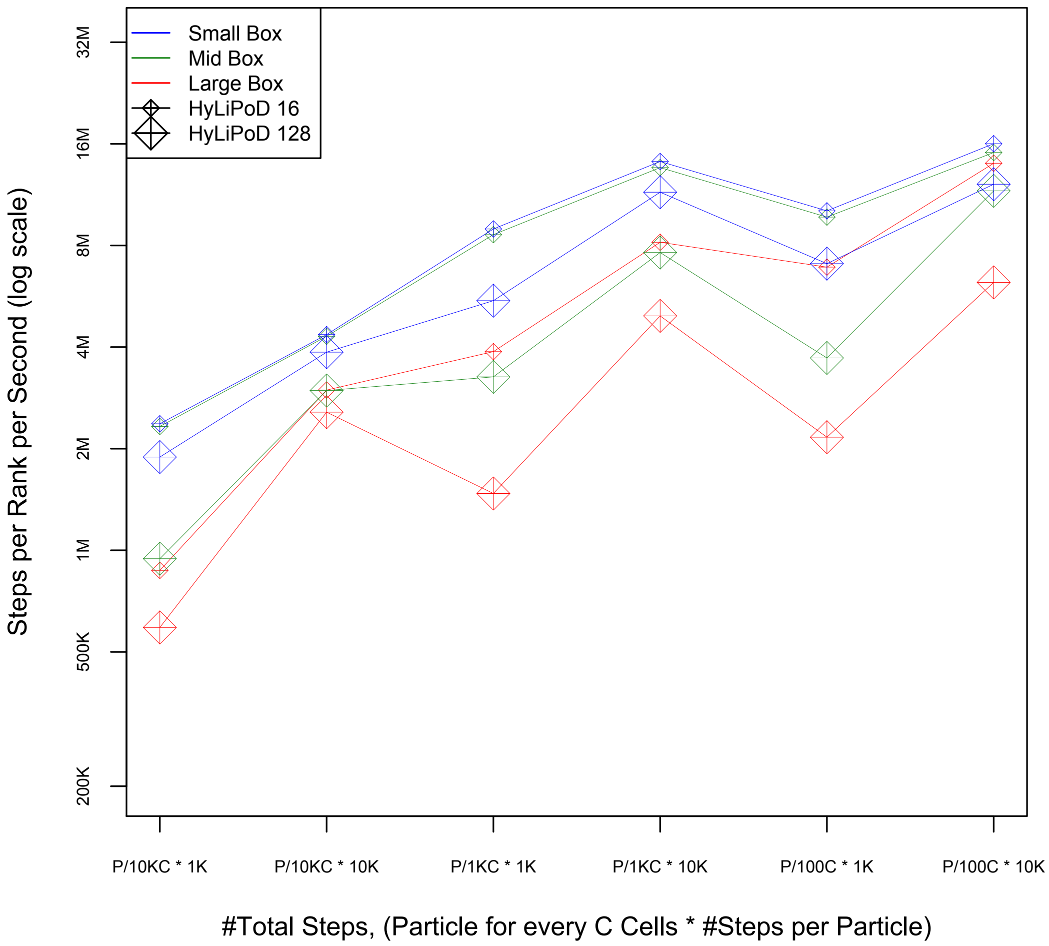


Figure 26. The performance scalability of our hybrid parallel particle advection algorithm (HyLiPoD) for different workloads.

which means less I/O operations or communication. The algorithm obtained the best performance using 16 ranks, $P/100C * 10K$ particles, where the algorithm reached a performance of 16 million $SPRPS$ for a small seeding box workload. The performance drops at larger scale since there are more blocks and particles causing additional I/O or communication cost.

Table 13. Comparing the performance of the largest workload of the three algorithms: HyLiPoD, LSM, and POD for different seeding box sizes.

Seeding box		HyLiPoD	LSM	POD
Large:	16Ranks	14M	10M	14M
	128Ranks	6M	6M	6M
Mid:	16Ranks	15M	15M	4M
	128Ranks	11M	11M	8K
Small:	16Ranks	16M	16M	1M
	128Ranks	12M	12M	6K

6.4.2 Comparing HyLiPoD to LSM and POD.

Table 13 compares the performance of the three algorithms: HyLiPoD, LSM, and POD for the largest workload ($P/100C * 10K$) for the three seeding box sizes. For the large seeding box, HyLiPoD performs similarly to POD since all the blocks are included. For the mid and small box sizes, HyLiPoD performs similarly to LSM since the number of blocks within the seeding box boundaries are less than 70% of the total number of blocks.

Figure 27 compares the performance of the three algorithms for all the different workloads. There are two observations that can be seen from the figure. The first one is that the HyLiPoD algorithm adapts its behavior depending on the workload. For the large seeding box workload HyLiPoD and POD have similar performances, while LSM has lower performance because of the I/O cost. On the other hand, for the mid and small seeding box workloads, HyLiPoD has a similar performance to LSM and POD has a lower performance due to the load imbalance. The second observation, is the performance drop down at large scale. The drop down at large scale is either because of additional I/O operations, additional communication, or increase in load imbalance. At the large box workload LSM has a higher drop down than the other two algorithms because the cost of I/O in LSM

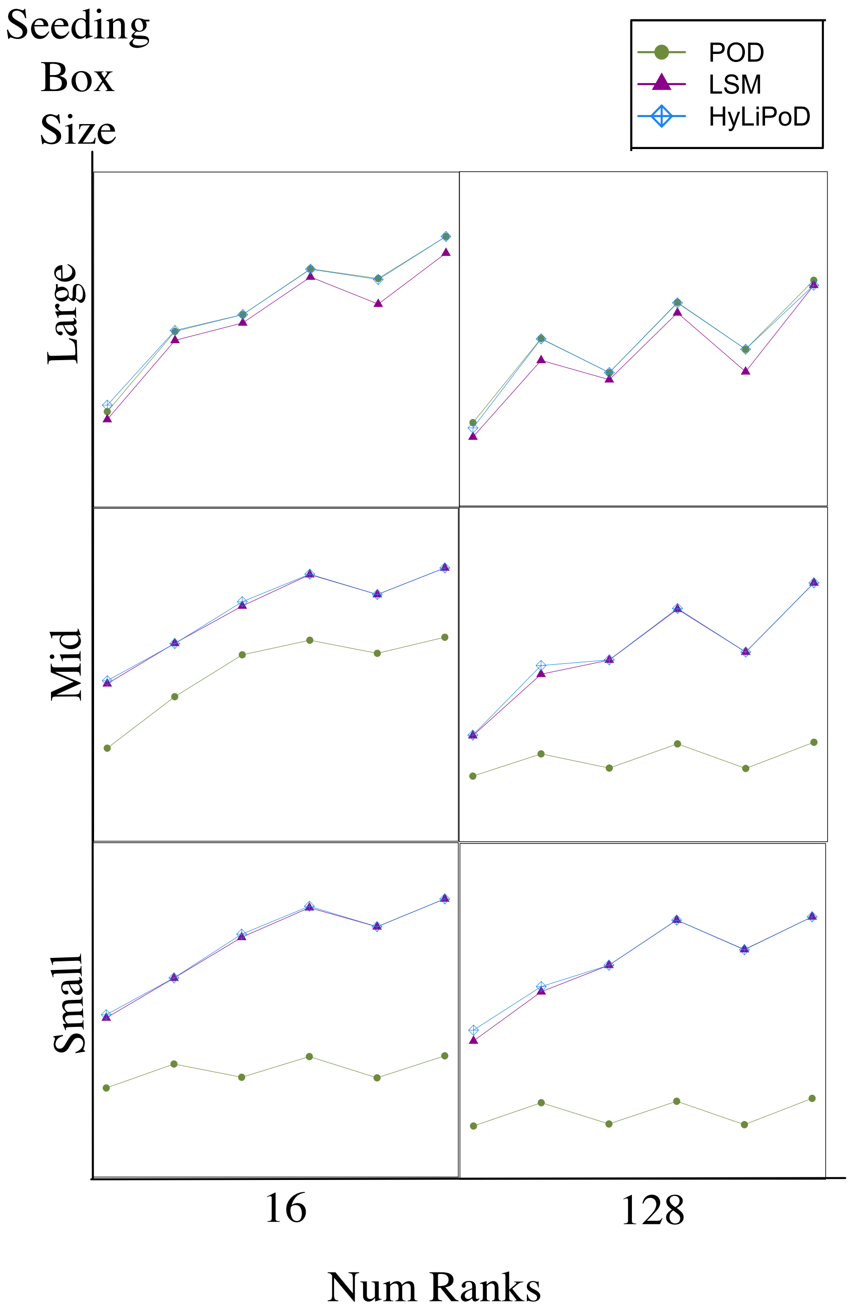


Figure 27. Comparing the performance of the three algorithms for different workloads.

is higher than the cost of communication in POD and HyLiPoD. In the case of mid and small box, POD has a higher drop down because the cost of the increased imbalance is higher than the cost of the additional I/O in LSM and HyLiPoD.

Part IV

The Future of Parallel Particle Advection

This part of the dissertation explores the furure of parallel particle advection. The first chapter studies the performance of the two traditional parallelization algorithms. The second chapter discusses the future work that can be done to improve parallel particle advection.

CHAPTER VII

IN SITU PARALLEL PARTICLE ADVECTION

Most of the text in this chapter comes from [9], which was a collaboration between David Pugmire (ORNL), Hank Childs (UO), and myself. This publication was written by myself and Hank Childs, with review and edits from David Pugmire. I was the main implementer of the software for this study, but used a code base that David Pugmire contributed to. Hank Childs assisted in analyzing results.

This chapter studies tightly-coupled in situ parallel particle advection by comparing the two traditional parallelization techniques for parallel particle advection. In a tightly-coupled in situ processing, the simulation code and the visualization routines are running on the same machine and sharing resources, where the simulation passes the data to the visualization routines. The visualization community has been using one of the traditional techniques (POD) due to its alignment with in situ constraints. This work explores whether other parallelization techniques are suitable for tightly-coupled in situ processing as well. Our findings demonstrate that parallelization techniques that have been used in post hoc research may still be relevant for in situ.

7.1 Motivation

Most of the work discussed in this dissertation, and over the last two decades on parallelizing particle advection algorithms, has come in the context of post hoc processing. In the post hoc setting, there is typically enough available memory for a given processing element (i.e., MPI task) to load multiple blocks, and also to store blocks redundantly across the processing elements. Further, acquiring a given block in a post hoc setting typically means reading it from disk, meaning that all blocks acquisitions (reads) take the same amount of time.

The assumptions made by post hoc algorithms change in a "tightly-coupled" in situ setting (i.e., where the simulation code and visualization routines share the same memory space). First, memory is assumed to be very precious because it is shared with the simulation code, which discourages acquiring multiple blocks and also having redundant blocks. Second, each processing element already has one block (i.e., the one the simulation code is operating on) and so the assumption is that the visualization routines should also operate on that same block, to save on memory. Finally, block acquisitions would no longer translate to reading data from disk, but instead acquiring data from another processing element via network communication.

Only one of the existing particle advection parallelization methods, POD (see Section 3.2.1), aligns with in situ constraints. In the tightly-coupled in situ setting, the block for a given processing element would be the same one the simulation code is operating on, minimizing memory usage.

The purpose of this work is to explore whether choices aside from POD are suitable for tightly-coupled in situ processing as well. While POD will minimize memory usage, it may be a poor choice with respect to execution time, which is also a very important consideration. In particular, POD performs poorly when particles are located in a small subset of the blocks, as this condition creates load imbalance.

To explore this theme, we introduce a straightforward variant of the POP algorithm (see Section 3.2.2) that is appropriate for in situ processing. The key difference between our in situ algorithm and the traditional (post hoc) POP algorithm is that in our algorithm a block is acquired via network communication from another processing element, while traditional POP acquires blocks from disk. In our experiments, we allowed each processing element to store up to two additional blocks (costing 40MB each), and found that runtimes improved by 10X over the POD algorithm for some workloads. While this additional memory overhead may be prohibitive in some settings, we feel our approach is useful in the settings where there is available memory.

Overall, we feel the main contribution of this work is to show that the wide body of previous research on parallelizing particle advection from the post hoc setting may still have a place in an in situ setting.

7.2 Related Work

Several works have employed particle advection techniques in situ. Most notably, Vetter et al. [138] presented an in situ framework for large unsteady flow data. Their solution used POD as a parallelization method. Further, an emerging in situ data reduction approach for vector fields uses parallel particle advection to calculate Lagrangian basis flows [139, 140, 141]. These works also use POD.

7.3 Algorithm

In this section, we present our POP implementation for an in situ context. For ease of reference, we abbreviate the term Processing Element as “PE” in our description. A PE equates to one MPI task. It also could equate to one compute node, provided there is one MPI task per node.

As discussed earlier, POP distributes particles across PEs, and the needed data blocks are acquired by each PE on demand. In a post hoc context, the data block is acquired by reading data from disk. To adapt the algorithm to work in an in situ framework, PEs in our algorithm acquire needed data blocks from other

PEs. We hypothesize that improvements in load balance will offset the cost of communicating data blocks, which can be high. Finally, our algorithm dedicates a separate thread for communication to hide the communication cost.

An important consideration for in situ POP is memory consumption. A PE acquiring many data blocks runs the risk of exceeding the budget allocated by the simulation for in situ processing. Instead, total memory needs to be controlled. Our algorithm allows the user to set the number of data blocks allowed in memory of a given PE. Before each data request, the algorithm checks if there is available space to make sure not to exceed the number of allowed data blocks. If the algorithm reached the maximum number of data blocks, it removes a block to make space for the new block. For our experiments, we set the maximum block size at two.

Algorithm 6 shows the pseudocode for the worker thread. It uses the following building blocks:

- Particle: a data structure that represents a particle in the vector field. The structure contains the particle id, position, current block id, and can also store the trajectory of the particle.
- ParticleArray: a data structure that stores an array of Particles.
- ArrayOfParticleArrays: a data structure that stores multiple elements of ParticleArray. Each of these elements stores multiple elements of Particle.
- SortParticleByBlock(): a function that sorts Particles depending on their current block id and returns two elements: ArrayOfParticleArrays and a vector containing the ids of needed blocks. All Particles that belongs to block i are stored in index i of ArrayOfParticleArrays.
- Advect(): a function that advects the Particles of a ParticleArray until they exit the current block or terminate. This function returns two ParticleArray

elements: the first one contains the completed particles, and the second one contains particles that need another data block.

- CheckForIncomingMessages(): a function that checks for incoming messages from other PEs. These messages can be data requests from other PEs or notifications of particle terminations.
- SendData(): a function that sends a data block to the requesting PE.
- RequestData(): a function that requests a data block from another PE.

The algorithm starts by distributing P particles across N PEs, assigning $\frac{P}{N}$ particles to each PE. Each PE then begins the process of advecting its particles. First, each PE starts by sorting particles by block and identifying the needed data blocks. Next, the worker thread advects the particles located in its local data block. We use the VTK-m [2] library for particle advection within a PE, specifically the module developed by Pugmire, et al. [3]. Simultaneous to advection, the communication thread requests needed data blocks; this is described in Algorithm 7. When a PE receives a requested data block, the PE's particles located in that data block would be advected. The algorithm completes when all particles are terminated, either by reaching the maximum advection step or exiting the problem domain.

An important consideration for our algorithm was the cost to send data. When a PE's block is requested, it employs a multi-threaded approach to serialize the data into a byte string. It also caches this byte string to prevent repeated serialization costs.

7.4 Experimental Overview

This section provides an overview of our experiments: experiment configurations (7.4.1) and the metrics we use to evaluate performance (4.2.4.4).

Algorithm 6 Pseudocode of the worker thread for one PE.

```
function IN-SITU-POP-ADVECT(ParticleArray pv)
    $keepGoing \leftarrow true$
    $ArrayOfParticleArrays\ \ pva[NUMBLOCKS]$
    $(pva, neededDataBlocks) \leftarrow SortParticlesByBlock(pv)$
    $allCompletedParticles \leftarrow \emptyset$
    while keepGoing do
        $contParticles \leftarrow \emptyset$
        for i in NUMBLOCKS do
            if pva[i].size() > 0 then
                $ParticleArray\ \ completed,\ \ continuing$
                $(completed, continuing) \leftarrow Advect(pva[i], b)$
                $allCompletedParticles\ \ += \ \ completed$
                $contParticles\ \ += \ \ continuing$
            end if
        end for
        if contParticles.size() > 0 then
            $pva \leftarrow SortParticlesByBlock(contParticles)$
        else
            $keepGoing \leftarrow false$
        end if
    end while
end function
```

7.4.1 Experiment Configurations.

7.4.1.1 Data Set:. Our study used an astrophysics data set consisting of 32 blocks, with each block containing 128^3 cells. It came from a simulation data of a magnetic field surrounding a solar core collapse, which results in a supernova. The simulation was performed via the GenASiS [134] code, which is a multi-physics code for astrophysical systems involving nuclear matter.

7.4.1.2 Level of concurrency:. We ran all experiments using 32 MPI tasks on 16 nodes of Cori, a machine at Lawrence Berkeley National Laboratory's NERSC facility. Cori has both Xeon Phi and Intel Xeon "Haswell" processor nodes; our experiments were run on the Haswells. We used 16 cores per MPI task, for a total of 512 cores in each run. We declined to use the hyper-

Algorithm 7 Pseudocode of the communication thread for one PE.

```
function In-Situ-POP-Communicate(int* neededDataBlocks)
    for i in neededDataBlocks do
        $owner \leftarrow GetOwnerNode(i)$
        $dataBuffer \leftarrow RequestData(owner, i)$
    end for
    if numActive > 0 then
        $keepCommunicating \leftarrow true$
    end if
    while keepCommunicating do
        $MSG \leftarrow CheckForIncomingMessages()$
        if MSG = PARTICLES_TERMINATED then
            numActive -= MSG.numTerminated
        else if MSG = NEED_DATA then
            $SendData(MSG.blockID)$
        end if
        if numActive < 0 then
            $keepCommunicating \leftarrow false$
        end if
    end while
end function
```

threading feature, since it did not boost performance for the VTK-m code base we were using. Each Haswell node on Cori has 128GB of memory.

7.4.1.3 Parallelization Techniques:. We consider both the POD algorithm and the POP extension we introduced in this study. The POP algorithm running on each PE was allowed to cache up to two blocks it acquired from other PEs. While in this study we limited the cache size to two additional blocks, the user can choose to increase the number of allowed data blocks in cache to improve performance but at the cost of a higher memory consumption. It is important to note that while our POP algorithm is designed for in situ, we ran in a so-called "theoretical" in situ environment, as our algorithm was not connected to a running simulation. Instead, before executing the algorithm, each PE acquired one block of data from disk. From this point forward, the disk was not consulted, and data

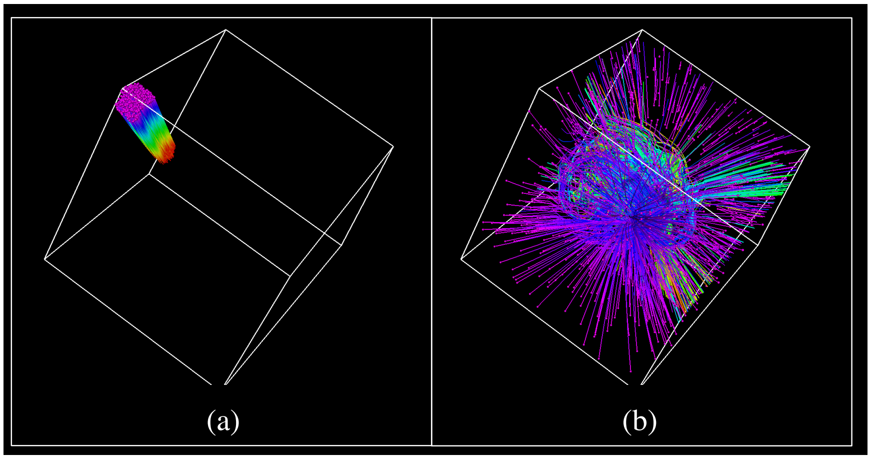
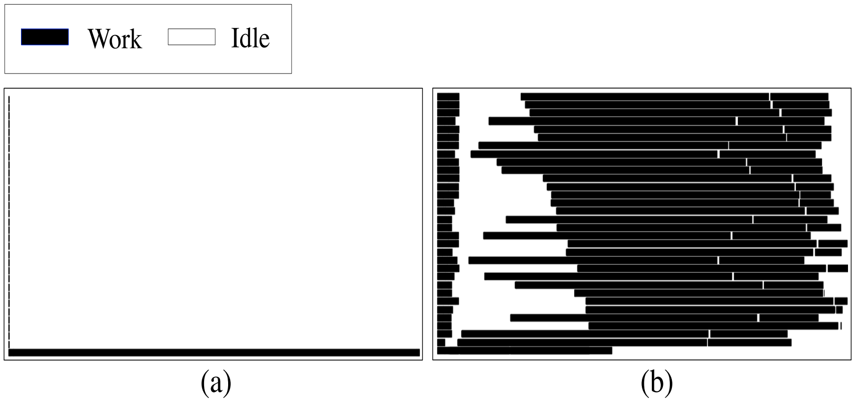


Figure 28. Streamlines visualization for our (a) dense and (b) uniform seed distributions.

was exchanged via network as it would be in an in situ setting. No I/O timings are reported, since we feel it is not relevant to our study.

7.4.1.4 Particle Workload:. We used one million particles, and advected each particle 10K steps (or fewer in the relatively rare cases where a particle exited the volume), for a total of approximately 10 billion advection steps. Particles were advected using velocity. We consider two extremes of seeding distributions: dense and uniform. In our study, the dense distribution was so concentrated that all of the particles begin in a single block, which is very likely to lead to load imbalance when using POD. Our uniform distribution had particles spread evenly throughout the volume, increasing communication cost when using POP, since more data blocks are required. Figure 28 shows a visualization of the two distributions.

7.4.2 Performance Measurement. For each phase, we display the execution time of the slowest PE, the maximum memory consumption needed to store the data, and the load imbalance. The load imbalance impacts the performance because the execution time is determined by the time of the slowest



(a) (b)

Figure 29. Performance of the two algorithms (a) POD, (b) POP, using 32 PEs to advect 1 million particles for 10 thousands steps for a dense distribution of seeds. The POD figure shows one task working the whole time (the task at the bottom), while the POP figure has more PEs involved. This figure is horizontally scaled based on run-time; POD ran for 307s, while POP ran for 26.6s.

PE. We define load imbalance with the following equation:

$$\text{Load imbalance} = \frac{\mathrm{T_s}}{\sum_{0<p<N} \mathrm{T_p}/\mathrm{N}}$$

where T_p is the total execution time for PE P, and T_s is the total execution time of the slowest PE.

7.5 Results

This section presents the results of our study. We divide our analysis based on the seed distribution: dense (7.5.1) and uniform (7.5.2).

7.5.1 Dense Distribution. The results for dense seeding are presented in Table 3. The results show that using POP improves performance by a factor of 11.5X over POD.

POD has a high execution time of 307s, due to the high load imbalance between PEs. This phenomenon is plotted in Figure 29. Since all particles are located in one data block (*block*0), there is one PE advecting all particles.

Table 14. Comparing the performance and memory consumption of the two algorithms for a dense particle distribution. Initialization time measures the time to initialize variables and generate initial seeds. Advection time measures the time to advect particles and to process the advection results (e.g., terminate). Communication time measures the time to request or send data blocks or particles to other PEs and to inform other PEs of termination. Idle time is the difference between total time and the sum of the other time measurements.

	POD	POP
Total time	307s	26.6s
Initialization time	0.97s	0.33s
Advection time	303s	16.9s
Communication time	0.18s	4.22s
Sort particle time	0.02s	0.1s
Load imbalance	30.34x	1.2x
Memory to store data	46.99MB	93.99 MB

Using POP distributes the workload and reduces the execution time to 26.6s. Even though the communication takes 4.2s, the overall execution time is lower than POD. As discussed in Section 7.3, we took care to optimize serialization time, which is an important component of communication time. We found that serializing a 128^3 data block took about one-eighth of a second. (Previous versions over our code that serialized with a single core were much slower.) Figure 29 shows that there is idle time for each PE after advecting the particles located in its block. This idle time is the time spent waiting to receive the required data block.

Using POP increases the memory requirement needed to store the data. This is because each PE is storing its data block and its received data blocks. In the case of dense distribution, only one extra block was needed, meaning that the cache of size two was only half-filled. The memory consumption presented in the table is representing the number of MB needed to store the velocity data; if a simulation code was calculating extra quantities (temperature, density, etc.), then the proportional increase in memory would be lower.

Table 15. Comparing the performance and memory consumption of the two algorithms for uniform seeding. The terms in this table are described in Table 1.

	POD	POP
Total time	23.9s	210s
Initialization time	0.65s	0.75s
Advection time	17.6s	99.7s
Communication time	4.34s	21.8s
Sort particle time	0.01s	13.9s
Load imbalance	1.19x	1.41x
Memory to store data	47.12MB	140.9MB

7.5.2 Uniform Distribution. The results for uniform seeding are presented in Table 15. With uniform seeding, POD performs 9.9X better than POP. This is because POP's PEs needed to request data blocks from the other 31 PEs, since its particles are scattered across the whole data domain. This leads to a higher communication time, in addition to idle time waiting for data blocks.

In this test, the PEs made use of both slots in its cache. This means at any given time, each PE could store a maximum of 140.9MB of vector field data. We anticipate that a larger cache could substantially reduce execution time. When the size is small, a smaller number of blocks can be requested at the same time, since our algorithm checks for available slots before each request. As a result, PEs might need to request the same data block more than once for cases where particles advect toward a previous data block.

7.6 Conclusion

The contribution of this chapter is an extension of the POP algorithm to work on an in situ context. We adapt the algorithm to acquire data blocks from other PEs instead of reading it from disk. The chapter compares between the main particle advection parallelization methods (POD and POP), and shows that our POP extension is superior for the workload where POD is known to perform poorly. Further, the study provides evidence that other parallelization techniques designed for post hoc processing may also be useful for in situ processing.

CHAPTER VIII

CONCLUSION AND FUTURE WORK

8.1 Synthesis

While there have been several solutions proposed to optimize parallel particle advection, there is a lack of understanding on how these parallel algorithms compare to each other. With this dissertation we gained a better understanding of the strengths and pitfalls of the different algorithms. The main two question of this dissertation are:

- **Which parallelization technique performs best for a given workload?**
- **What are the unsolved problems in parallel particle advection? Are there any workloads that are difficult to balance using existing parallelization techniques?**

To answer these questions, we improved and adopted the best practices for the individual algorithms. For all algorithms we added on node parallelism by using the VTK-m library. We also implemented a cache mechanism based on a previous research for all the algorithms loading data on demand. We improved the work requesting algorithm by incorporating a new scheduling method (Lifeline). Our results showed significant improvements over previous implementations. Then we performed a parallel particle advection bake-off study (Chapter V) to understand the behavior of these algorithms on different workloads. In this study we compared four of the most popular parallel particle advection solutions and ran experiments over various workloads. The results we got from our study answer our dissertation questions.

- **Which parallelization technique performs best for a given workload?**

  Our experiments showed that the seeding box is a major consideration when considering which algorithm to choose. For a small or medium size seeding box, the LSM algorithm performed the best, while the POD method performed the best for a large seeding box. These results hold true for different number of particles, different number of maximum steps, and at different scales.

- **What are the unsolved problems in parallel particle advection? Are there any workloads that are difficult to balance using existing parallelization techniques?**

  The performance of each of these algorithms decreases as the scale increases due to the increased number of communications, I/O operations, and more severe load imbalance. In addition, all algorithms perform poorly for the workload with the fewest particle advection steps since there is not enough work to offset the cost of I/O and communication.

The results of the bake-off study showed that the two algorithms that performed the best are POD and LSM algorithms. Each of these two algorithms performed the best for different workloads, while still suffering poor performance when the other algorithm performs best. To address these individual weaknesses, we implemented a new algorithm (HyLiPod) that is a hybrid between the POD and LSM algorithms. Our algorithm adapted its behavior depending on the workload characteristics and applied the algorithm that achieves the best performance.

Traditional visualization is usually performed post hoc, meaning saving the data and performing the visualization after the simulation is completed. However, with the increasing gap between the computational power and I/O capabilities, saving data to disk is becoming a bottleneck. In situ visualization is a promising solution to reduce the cost of I/O by visualizing the simulation as it is running, avoiding intermediate data files. Previous in situ parallel particle advection solutions used POD. While POD aligns with in situ constrains it can lead to load imbalance for some workloads. We adapted the POP algorithm to work in an in situ setting and we compared the performance of POD and POP for a dense and sparse seed distribution. Our study demonstrated that other parallel particle advection algorithms might be suitable for in situ setting.

8.2 Recommendations for Future Study

8.2.1 A Scalable Parallel Particle Advection Algorithm.

The results of the bake-off study showed that the performance of all algorithms drops at scale. Even though HyLiPod has the best performance since it avoids the weakness that other algorithms have for specific workloads, it still does not scale well. This area represents future research. One of the reasons for poor performance is the communication cost. An interesting study would be to evaluate the impact of adapting the application communication pattern to take into account the network topology [142, 143].

8.2.2 Integrating Into Production Visualization Tools.

Integrating our current implementation to the VTK-h library [144] will help visualization users to use different parallel particle advection algorithms for their workloads and help the visualization community to continue this research. Our implementation supports different factors, which allows the user to test several workloads. It can also be extended to support new parallel particle advection algorithms. To this end, we plan to integrate our algorithms from Chapter VII with Ascent [144].

8.2.3 In Situ Parallel Particle Advection.

Having a single platform to study the different parallel particle advection algorithms helped to understand the strength and weaknesses of each of these algorithms. Future work should study these algorithms in an in situ context, since our study in Chapter VII showed that adapting some of the other parallel algorithms to work in situ can lead to major performance improvements for some of the workloads.

8.2.4 Architecture Related Considerations.

Our bake-off study studied showed that the most important factors impacting the different algorithms performance are their ability to load balance the workloads and their communication and I/O costs. While different architectures may slightly impact the cost of communications or I/O, the overall strength and weakness of each algorithm would remain the same. For instance, the POD algorithm would still have the best performance for a large box workload since the cost of communication is less than the cost of I/O The POD would still have bad performance for a small seeding box, regardless of any architecture change, as the load imbalance is the major factor for such performance drop. Overall, a change in architecture would not change the way an algorithm load balance its workload or an algorithm communication and I/O patterns. Therefore, the lessons we learned from the bake-off study still hold true for different architectures. Regardless, a future study of different architectures would reveal the magnitude of effects from changes in compute and communication.

8.2.5 Performance Model for Parallel Particle Advection. The lessons learned from the bake-off study helped us to determine the important factors impacting the performance. We could use these factors to create a performance model for each individual parallel algorithm to estimate the performance of the algorithm. That information would assist in choosing the best parallel algorithm for the given workload.

REFERENCES CITED

[1] P. Prince and J. Dormand, "High order embedded runge-kutta formulae," *Journal of Computational and Applied Mathematics*, vol. 7, no. 1, pp. 67 – 75, 1981.

[2] K. Moreland, C. Sewell, W. Usher, L. 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 (CG&A)*, vol. 36, pp. 48–58, May/June 2016.

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

[4] R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing," *J. ACM*, vol. 46, pp. 720–748, Sept. 1999.

[5] V. A. Saraswat, P. Kambadur, S. Kodali, D. Grove, and S. Krishnamoorthy, "Lifeline-based global load balancing," in *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11, (New York, NY, USA), pp. 201–212, ACM, 2011.

[6] W. Zhang, O. Tardieu, D. Grove, B. Herta, T. Kamada, V. A. Saraswat, and M. Takeuchi, "Glb: lifeline-based global load balancing library in x10," in *PPAA@PPoPP*, 2014.

[7] V. Kumar, K. Murthy, V. Sarkar, and Y. Zheng, "Optimized distributed work-stealing," in *Proceedings of the Sixth Workshop on Irregular Applications: Architectures and Algorithms*, (Piscataway, NJ, USA), pp. 74–77, IEEE Press, 2016.

[8] R. Binyahib, D. Pugmire, B. Norris, and H. Childs, "A lifeline-based approach for work requesting and parallel particle advection," in *2019 IEEE 9th Symposium on Large Data Analysis and Visualization (LDAV)*, pp. 52–61, Oct 2019.

[9] 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*, ISAV '19, (New York, NY, USA), p. 29–33, Association for Computing Machinery, 2019.

[10] H. Childs, D. Pugmire, S. Ahern, B. Whitlock, M. Howison, , G. H. Weber, and E. W. Bethel, “Extreme scaling of production visualization software on diverse architectures,” *IEEE Computer Graphics and Applications*, vol. 30, pp. 22–31, May 2010.

[11] T. Peterka, H. Yu, R. Ross, K. Ma, and R. Latham, “End-to-end study of parallel volume rendering on the ibm blue gene/p,” in *2009 International Conference on Parallel Processing*, pp. 566–573, Sept 2009.

[12] J. Clyne, P. D. Mininni, A. Norton, and M. Rast, “Interactive desktop analysis of high resolution simulations : application to turbulent plume dynamics and current sheet formation,” 2007.

[13] V. Pascucci and R. J. Frank, “Global static indexing for real-time exploration of very large regular grids,” in *SC ’01: Proceedings of the 2001 ACM/IEEE Conference on Supercomputing*, pp. 45–45, Nov 2001.

[14] H. Childs, E. Brugger, K. Bonnell, J. Meredith, M. Miller, B. Whitlock, and N. Max, “A contract based system for large data visualization,” in *VIS 05. IEEE Visualization, 2005.*, pp. 191–198, Oct 2005.

[15] O. Rubel, , , H. Childs, J. Meredith, C. G. R. Geddes, E. Cormier-Michel, S. Ahern, G. H. Weber, P. Messmer, H. Hagen, B. Hamann, and E. Wes Bethel, “High performance multivariate visual data exploration for extremely large data,” in *SC ’08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, pp. 1–12, Nov 2008.

[16] A. C. Bauer, H. Abbasi, J. Ahrens, H. Childs, B. Geveci, S. Klasky, K. Moreland, P. O’Leary, V. Vishwanath, B. Whitlock, and E. W. Bethel, “In situ methods, infrastructures, and applications on high performance computing platforms,” in *Proceedings of the Eurographics / IEEE VGTC Conference on Visualization: State of the Art Reports*, EuroVis ’16, (Goslar Germany, Germany), pp. 577–597, Eurographics Association, 2016.

[17] N. Fabian, K. Moreland, D. Thompson, A. C. Bauer, P. Marion, B. Gevecik, M. Rasquin, and K. E. Jansen, “The paraview coprocessing library: A scalable, general purpose in situ visualization library,” in *2011 IEEE Symposium on Large Data Analysis and Visualization*, pp. 89–96, Oct 2011.

[18] B. Whitlock, J. M. Favre, and J. S. Meredith, “Parallel In Situ Coupling of Simulation with a Fully Featured Visualization System,” in *Eurographics Symposium on Parallel Graphics and Visualization* (T. Kuhlen, R. Pajarola, and K. Zhou, eds.), The Eurographics Association, 2011.

[19] W. J. Schroeder, K. M. Martin, and W. E. Lorensen, "The design and implementation of an object-oriented toolkit for 3D graphics and visualization," in *VIS '96: Proceedings of the 7th conference on Visualization '96*, pp. 93–ff., IEEE Computer Society Press, 1996.

[20] C. Upson, T. F. Jr., D. Kamins, D. H. Laidlaw, D. Schlegel, J. Vroom, R. Gurwitz, and A. van Dam, "The application visualization system: A computational environment for scientific visualization," *Computer Graphics and Applications*, vol. 9, pp. 30–42, July 1989.

[21] S. Grottel, M. Krone, C. Müller, G. Reina, and T. Ertl, "Megamol: A prototyping framework for particle-based visualization," *IEEE Transactions on Visualization and Computer Graphics*, vol. 21, pp. 201–214, Feb 2015.

[22] H. Childs, E. Brugger, B. Whitlock, J. Meredith, S. Ahern, D. Pugmire, K. Biagas, M. Miller, C. Harrison, G. H. Weber, H. Krishnan, T. Fogal, A. Sanderson, C. Garth, E. W. Bethel, D. Camp, O. Rübel, M. Durant, J. M. Favre, and P. Navrátil, "VisIt: An End-User Tool For Visualizing and Analyzing Very Large Data," in *High Performance Visualization–Enabling Extreme-Scale Scientific Insight*, pp. 357–372, Oct 2012.

[23] J. Ahrens, B. Geveci, C. Law, C. Hansen, and C. Johnson, "36-paraview: An end-user tool for large-data visualization," *The visualization handbook*, vol. 717, 2005.

[24] B. S. Siegell and P. Steenkiste, "Automatic generation of parallel programs with dynamic load balancing," in *Proceedings of 3rd IEEE International Symposium on High Performance Distributed Computing*, pp. 166–175, Aug 1994.

[25] M. Woo and O. A. R. Board, *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 1.2*. Graphics programming, Addison-Wesley, 1999.

[26] P. Shirley, M. Ashikhmin, and S. Marschner, *Fundamentals of Computer Graphics*. Ak Peters Series, Taylor & Francis, 2005.

[27] R. A. Drebin, L. Carpenter, and P. Hanrahan, "Volume rendering," in *Proceedings of the 15th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '88, (New York, NY, USA), pp. 65–74, ACM, 1988.

[28] N. Max, "Optical models for direct volume rendering," *IEEE Transactions on Visualization and Computer Graphics*, vol. 1, pp. 99–108, June 1995.

[29] M. Hadwiger, J. M. Kniss, C. Rezk-salama, D. Weiskopf, and K. Engel, *Real-time Volume Graphics*. Natick, MA, USA: A. K. Peters, Ltd., 2006.

[30] S. Molnar, M. Cox, D. Ellsworth, and H. Fuchs, "A sorting classification of parallel rendering," *IEEE Computer Graphics and Applications*, vol. 14, pp. 23–32, July 1994.

[31] R. Samanta, J. Zheng, T. Funkhouser, K. Li, and J. P. Singh, "Load balancing for multi-projector rendering systems," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware*, HWWS '99, (New York, NY, USA), pp. 107–116, ACM, 1999.

[32] F. Erol, S. Eilemann, and R. Pajarola, "Cross-Segment Load Balancing in Parallel Rendering," in *Eurographics Symposium on Parallel Graphics and Visualization* (T. Kuhlen, R. Pajarola, and K. Zhou, eds.), The Eurographics Association, 2011.

[33] B. Moloney, M. Ament, D. Weiskopf, and T. Moller, "Sort-first parallel volume rendering," *IEEE Transactions on Visualization and Computer Graphics*, vol. 17, pp. 1164–1177, Aug 2011.

[34] E. W. Bethel, H. Childs, and C. Hansen, *High Performance Visualization: Enabling Extreme-Scale Scientific Insight*. Chapman & Hall/CRC, 1st ed., 2012.

[35] S. Marchesin, C. Mongenet, and J.-M. Dischler, "Dynamic load balancing for parallel volume rendering," in *Proceedings of the 6th Eurographics Conference on Parallel Graphics and Visualization*, EGPGV '06, (Aire-la-Ville, Switzerland, Switzerland), pp. 43–50, Eurographics Association, 2006.

[36] K.-L. Ma, J. S. Painter, C. D. Hansen, and M. F. Krogh, "Parallel volume rendering using binary-swap compositing," *IEEE Computer Graphics and Applications*, vol. 14, pp. 59–68, July 1994.

[37] K.-L. Ma and T. W. Crockett, "A scalable parallel cell-projection volume rendering algorithm for three-dimensional unstructured data," in *In Proceedings of 1997 Symposium on Parallel Rendering*, pp. 95–104.

[38] D. Steiner, E. G. Paredes, S. Eilemann, and R. Pajarola, "Dynamic work packages in parallel rendering," in *Proceedings of the 16th Eurographics Symposium on Parallel Graphics and Visualization*, EGPGV '16, (Goslar Germany, Germany), pp. 89–98, Eurographics Association, 2016.

[39] S. Eilemann, M. Makhinya, and R. Pajarola, "Equalizer: A scalable parallel rendering framework," *IEEE Transactions on Visualization and Computer Graphics*, vol. 15, pp. 436–452, May 2009.

[40] C. Müller, M. Strengert, and T. Ertl, "Optimized volume raycasting for graphics-hardware-based cluster systems," in *Proceedings of the 6th Eurographics Conference on Parallel Graphics and Visualization*, EGPGV '06, (Aire-la-Ville, Switzerland, Switzerland), pp. 59–67, Eurographics Association, 2006.

[41] C. Montani, R. Perego, and R. Scopigno, "Parallel rendering of volumetric data set on distributed-memory architectures," *Concurrency: Practice and Experience*, vol. 5, no. 2, pp. 153–167, 1993.

[42] H. Childs, M. Duchaineau, and K.-L. Ma, "A scalable, hybrid scheme for volume rendering massive data sets," pp. 153–161, 2006.

[43] R. Binyahib, T. Peterka, M. Larsen, K. Ma, and H. Childs, "A scalable hybrid scheme for ray-casting of unstructured volume data," *IEEE Transactions on Visualization and Computer Graphics*, pp. 1–1, 2018.

[44] R. Samanta, T. Funkhouser, K. Li, and J. P. Singh, "Hybrid sort-first and sort-last parallel rendering with a cluster of pcs," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware*, HWWS '00, (New York, NY, USA), pp. 97–108, ACM, 2000.

[45] A. Garcia and H.-W. Shen, "An interleaved parallel volume renderer with pc-clusters," in *Proceedings of the Fourth Eurographics Workshop on Parallel Graphics and Visualization*, EGPGV '02, (Aire-la-Ville, Switzerland, Switzerland), pp. 51–59, Eurographics Association, 2002.

[46] K.-L. Ma, "Parallel volume ray-casting for unstructured-grid data on distributed-memory architectures," in *Proceedings of the IEEE Symposium on Parallel Rendering*, PRS '95, (New York, NY, USA), pp. 23–30, ACM, 1995.

[47] N. Max, P. Williams, C. Silva, and R. Cook, "Volume rendering for curvilinear and unstructured grids," in *Computer Graphics International, 2003. Proceedings*, pp. 210–215, IEEE, 2003.

[48] M. Larsen, S. Labasan, P. Navrátil, J. Meredith, and H. Childs, "Volume Rendering Via Data-Parallel Primitives," in *Proceedings of EuroGraphics Symposium on Parallel Graphics and Visualization (EGPGV)*, (Cagliari, Italy), pp. 53–62, May 2015.

[49] W. E. Lorensen and H. E. Cline, "Marching cubes: A high resolution 3d surface construction algorithm," in *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '87, (New York, NY, USA), pp. 163–169, ACM, 1987.

[50] Y.-J. Chiang, R. Farias, C. T. Silva, and B. Wei, "A unified infrastructure for parallel out-of-core isosurface extraction and volume rendering of unstructured grids," in *Proceedings IEEE 2001 Symposium on Parallel and Large-Data Visualization and Graphics (Cat. No.01EX520)*, pp. 59–151, Oct 2001.

[51] Y.-J. Chiang, C. T. Silva, and W. J. Schroeder, "Interactive out-of-core isosurface extraction," in *Proceedings of the Conference on Visualization '98*, VIS '98, (Los Alamitos, CA, USA), pp. 167–174, IEEE Computer Society Press, 1998.

[52] M. J. Quinn, *Designing Efficient Algorithms for Parallel Computers.* New York, NY, USA: McGraw-Hill, Inc., 1987.

[53] X. Zhang, C. Bajaj, and W. Blanke, "Scalable isosurface visualization of massive datasets on cots clusters," in *Proceedings of the IEEE 2001 Symposium on Parallel and Large-data Visualization and Graphics*, PVG '01, (Piscataway, NJ, USA), pp. 51–58, IEEE Press, 2001.

[54] C. L. Bajaj, V. Pascucci, and D. R. Schikore, "The contour spectrum," in *Proceedings. Visualization '97 (Cat. No. 97CB36155)*, pp. 167–173, Oct 1997.

[55] L. Arge and J. S. Vitter, "Optimal dynamic interval management in external memory (extended abstract)," in *FOCS*, 1996.

[56] M. Berger and P. Colella, "Local adaptive mesh refinement for shock hydrodynamics," *Journal of Computational Physics*, vol. 82, no. 1, pp. 64 – 84, 1989.

[57] D. C. Fang, G. H. Weber, H. Childs, E. S. Brugger, B. Hamann, and K. I. Joy, "Extracting geometrically continuous isosurfaces from adaptive mesh refinement data," in *Proceedings of 2004 Hawaii International Conference on Computer Sciences*, pp. 216–224, 2004.

[58] G. H. Weber, H. Childs, and J. S. Meredith, "Efficient parallel extraction of crack-free isosurfaces from adaptive mesh refinement (amr) data," in *IEEE Symposium on Large Data Analysis and Visualization (LDAV)*, pp. 31–38, Oct 2012.

[59] G. H. Weber, O. Kreylos, T. J. Ligocki, J. Shalf, H. Hagen, B. Hamann, and K. I. Joy, "Extraction of crack-free isosurfaces from adaptive mesh refinement data," in *VisSym*, 2001.

[60] S. Eilemann and R. Pajarola, “Direct send compositing for parallel sort-last rendering,” in *Proceedings of the 7th Eurographics Conference on Parallel Graphics and Visualization*, EGPGV ’07, (Aire-la-Ville, Switzerland, Switzerland), pp. 29–36, Eurographics Association, 2007.

[61] H. Yu, C. Wang, and K.-L. Ma, “Massively parallel volume rendering using 2-3 swap image compositing,” in *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, SC ’08, (Piscataway, NJ, USA), pp. 48:1–48:11, IEEE Press, 2008.

[62] J. Nonaka, K. Ono, and M. Fujita, “234 scheduling of 3-2 and 2-1 eliminations for parallel image compositing using non-power-of-two number of processes,” in *2015 International Conference on High Performance Computing Simulation (HPCS)*, pp. 421–428, July 2015.

[63] J. Nonaka, K. Ono, and M. Fujita, “234compositor: A flexible parallel image compositing framework for massively parallel visualization environments,” *Future Generation Computer Systems*, vol. 82, pp. 647 – 655, 2018.

[64] R. Rabenseifner and J. L. Träff, “More efficient reduction algorithms for non-power-of-two number of processors in message-passing parallel systems,” in *Recent Advances in Parallel Virtual Machine and Message Passing Interface* (D. Kranzlmüller, P. Kacsuk, and J. Dongarra, eds.), (Berlin, Heidelberg), pp. 36–46, Springer Berlin Heidelberg, 2004.

[65] T. Peterka, D. Goodell, R. Ross, H. W. Shen, and R. Thakur, “A configurable algorithm for parallel image-compositing applications,” in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pp. 1–10, Nov 2009.

[66] K. Moreland, W. Kendall, T. Peterka, and J. Huang, “An image compositing solution at scale,” in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC ’11, (New York, NY, USA), pp. 25:1–25:10, ACM, 2011.

[67] J. Ahrens and J. Painter, “Efficient sort-last rendering using compression-based image compositing,” in *in Proceedings of the 2nd Eurographics Workshop on Parallel Graphics and Visualization*, pp. 145–151, 1998.

[68] K. Moreland, B. Wylie, and C. Pavlakos, “Sort-last parallel rendering for viewing extremely large data sets on tile displays,” in *Proceedings of the IEEE 2001 Symposium on Parallel and Large-data Visualization and Graphics*, PVG ’01, (Piscataway, NJ, USA), pp. 85–92, IEEE Press, 2001.

[69] D.-L. Yang, J.-C. Yu, and Y.-C. Chung, "Efficient compositing methods for the sort-last-sparse parallel volume rendering system on distributed memory multicomputers," in *Proceedings of the 1999 International Conference on Parallel Processing*, pp. 200–207, Sep. 1999.

[70] A. Takeuchi, F. Ino, and K. Hagihara, "An improvement on binary-swap compositing for sort-last parallel rendering," in *Proceedings of the 2003 ACM Symposium on Applied Computing*, SAC '03, (New York, NY, USA), pp. 996–1002, ACM, 2003.

[71] W. Kendall, T. Peterka, J. Huang, H.-W. Shen, and R. Ross, "Accelerating and benchmarking radix-k image compositing at large scale," in *Proceedings of the 10th Eurographics Conference on Parallel Graphics and Visualization*, EG PGV'10, (Aire-la-Ville, Switzerland, Switzerland), pp. 101–110, Eurographics Association, 2010.

[72] K. Moreland, "Icet users' guide and reference, version 2.0. technical report sand2010-7451, sandia national laboratories," January 2011.

[73] K. Moreland, "Comparing binary-swap algorithms for odd factors of processes," in *2018 IEEE 8th Symposium on Large Data Analysis and Visualization (LDAV)*, Oct 2018.

[74] M. Isenburg, P. Lindstrom, and H. Childs, "Parallel and streaming generation of ghost data for structured grids," *IEEE Computer Graphics and Applications*, vol. 30, pp. 32–44, May 2010.

[75] F. B. Kjolstad and M. Snir, "Ghost cell pattern," in *Proceedings of the 2010 Workshop on Parallel Programming Patterns*, ParaPLoP '10, (New York, NY, USA), pp. 4:1–4:9, ACM, 2010.

[76] C. R. Johnson, S. G. Parker, and D. Weinstein, "Large-scale computational science applications using the scirun problem solving environment," in *In Supercomputer*, 2000.

[77] 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*, vol. 24, pp. 954–963, Jan 2018.

[78] C. Harrison, J. Weiler, R. Bleile, K. Gaither, and H. Childs, "A distributed-memory algorithm for connected components labeling of simulation data," in *Topological and Statistical Methods for Complex Data* (J. Bennett, F. Vivodtzev, and V. Pascucci, eds.), (Berlin, Heidelberg), pp. 3–19, Springer Berlin Heidelberg, 2015.

[79] K. Moreland, L. Avila, and L. A. Fisk, “Parallel unstructured volume rendering in ParaView,” in *Visualization and Data Analysis 2007* (R. F. Erbacher, J. C. Roberts, M. T. Gröhn, and K. Börner, eds.), vol. 6495, pp. 144 – 155, International Society for Optics and Photonics, SPIE, 2007.

[80] J. M. Patchett, B. Nouanesengesy, J. Pouderoux, J. Ahrens, and H. Hagen, “Parallel multi-layer ghost cell generation for distributed unstructured grids,” in *2017 IEEE 7th Symposium on Large Data Analysis and Visualization (LDAV)*, pp. 84–91, Oct 2017.

[81] J. Biddiscombe, “High-Performance Mesh Partitioning and Ghost Cell Generation for Visualization Software,” in *Eurographics Symposium on Parallel Graphics and Visualization* (E. Gobbetti and W. Bethel, eds.), The Eurographics Association, 2016.

[82] E. G. Boman, U. V. Catalyurek, C. Chevalier, and K. D. Devine, “The Zoltan and Isorropia parallel toolkits for combinatorial scientific computing: Partitioning, ordering, and coloring,” *Scientific Programming*, vol. 20, no. 2, pp. 129–150, 2012.

[83] K. Moreland, “A survey of visualization pipelines,” *IEEE Transactions on Visualization and Computer Graphics*, vol. 19, pp. 367–378, March 2013.

[84] 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*, vol. 21, pp. 34–41, July 2001.

[85] B. Whitlock, J. M. Favre, and J. S. Meredith, “Parallel in situ coupling of simulation with a fully featured visualization system,” in *Proceedings of the 11th Eurographics Conference on Parallel Graphics and Visualization*, EGPGV ’11, (Aire-la-Ville, Switzerland, Switzerland), pp. 101–109, Eurographics Association, 2011.

[86] C. Mommessin, M. Dreher, B. Raffin, and T. Peterka, “Automatic data filtering for in situ workflows,” in *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 370–378, Sep. 2017.

[87] T. Peterka, J. Kwan, A. Pope, H. Finkel, K. Heitmann, and S. Habib, “Meshing the universe : Identifying voids in cosmological simulations through in situ parallel voronoi tessellation,” 2012.

[88] T. Peterka, D. Morozov, and C. Phillips, “High-performance computation of distributed-memory parallel 3d voronoi and delaunay tessellation,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC ’14, (Piscataway, NJ, USA), pp. 997–1007, IEEE Press, 2014.

[89] L. J. Guibas, D. E. Knuth, and M. Sharir, "Randomized incremental construction of delaunay and voronoi diagrams," *Algorithmica*, vol. 7, pp. 381–413, Jun 1992.

[90] C. B. Barber, D. P. Dobkin, D. P. Dobkin, and H. Huhdanpaa, "The quickhull algorithm for convex hulls," *ACM Trans. Math. Softw.*, vol. 22, pp. 469–483, Dec. 1996.

[91] A. Fabri and S. Pion, "Cgal - the computational geometry algorithms library," pp. 538–539, 01 2009.

[92] D. Morozov and T. Peterka, "Efficient delaunay tessellation through k-d tree decomposition," in *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 728–738, Nov 2016.

[93] A. Aggarwal and S. Vitter, Jeffrey, "The input/output complexity of sorting and related problems," *Commun. ACM*, vol. 31, pp. 1116–1127, Sept. 1988.

[94] J. S. Vitter, "External memory algorithms and data structures: Dealing with massive data," *ACM Comput. Surv.*, vol. 33, pp. 209–271, June 2001.

[95] Y.-J. Chiang and C. T. Silva in *External Memory Algorithms* (J. M. Abello and J. S. Vitter, eds.), ch. External Memory Techniques for Isosurface Extraction in Scientific Visualization, pp. 247–277, Boston, MA, USA: American Mathematical Society, 1999.

[96] C. Chen, L. Xu, T. Lee, and H. Shen, "A flow-guided file layout for out-of-core streamline computation," in *2011 IEEE Symposium on Large Data Analysis and Visualization*, pp. 115–116, Oct 2011.

[97] C. Chen, B. Nouanesengsy, T. Lee, and H. Shen, "Flow-guided file layout for out-of-core pathline computation," in *IEEE Symposium on Large Data Analysis and Visualization (LDAV)*, pp. 109–112, Oct 2012.

[98] C. Chen and H. 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)*, pp. 15–23, Oct 2013.

[99] L. Ibanez, W. Schroeder, L. Ng, and J. Cates, "Streaming large data. in the itk software guide," 2005.

[100] D. Nagle, "Mpi – the complete reference, vol. 1, the mpi core, 2nd ed., scientific and engineering computation series, by marc snir, steve otto, steven huss-lederman, david walker and jack dongarra," *Sci. Program.*, vol. 13, pp. 57–63, Jan. 2005.

[101] B. Chapman, G. Jost, R. van der Pas, and D. Kuck, *Using OpenMP: Portable Shared Memory Parallel Programming.* No. v. 10 in Scientific Computation Series, Books24x7.com, 2008.

[102] B. Nichols, D. Buttlar, J. Farrell, and J. Farrell, *PThreads Programming: A POSIX Standard for Better Multiprocessing.* A POSIX standard for better multiprocessing, O'Reilly Media, Incorporated, 1996.

[103] M. Howison, E. W. Bethel, and H. Childs, "Hybrid parallelism for volume rendering on large-, multi-, and many-core systems," *IEEE Transactions on Visualization and Computer Graphics*, vol. 18, pp. 17–29, Jan 2012.

[104] M. Howison, E. W. Bethel, and H. Childs, "Mpi-hybrid parallelism for volume rendering on large, multi-core systems," in *Proceedings of the 10th Eurographics Conference on Parallel Graphics and Visualization*, EG PGV'10, (Aire-la-Ville, Switzerland, Switzerland), pp. 1–10, Eurographics Association, 2010.

[105] 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*, pp. 1–12, Nov 2009.

[106] D. Camp, C. Garth, H. Childs, D. Pugmire, and K. Joy, "Streamline integration using mpi-hybrid parallelism on a large multicore architecture," *IEEE Transactions on Visualization and Computer Graphics*, vol. 17, pp. 1702–1713, Nov 2011.

[107] C. Garth, F. Gerhardt, X. Tricoche, and H. Hans, "Efficient computation and visualization of coherent structures in fluid flow applications," *IEEE Transactions on Visualization and Computer Graphics*, vol. 13, pp. 1464–1471, Nov. 2007.

[108] J. P. M. Hultquist, "Constructing stream surfaces in steady 3d vector fields," in *Visualization, 1992. Visualization '92, Proceedings., IEEE Conference on*, pp. 171–178, Oct 1992.

[109] C. Garth, X. Tricoche, T. Salzbrunn, T. Bobach, and G. Scheuermann, "Surface techniques for vortex visualization," in *Proceedings of the Sixth Joint Eurographics - IEEE TCVG Conference on Visualization*, VISSYM'04, (Aire-la-Ville, Switzerland, Switzerland), pp. 155–164, Eurographics Association, 2004.

[110] T. McLoughlin, R. S. Laramee, and E. Zhang, “Easy integral surfaces: A fast, quad-based stream and path surface algorithm,” in *Proceedings of the 2009 Computer Graphics International Conference*, CGI ’09, (New York, NY, USA), pp. 73–82, ACM, 2009.

[111] H. Krishnan, C. Garth, and K. Joy, “Time and streak surfaces for flow visualization in large time-varying data sets,” *IEEE Transactions on Visualization and Computer Graphics*, vol. 15, pp. 1267–1274, Nov 2009.

[112] T. McLoughlin, R. S. Laramee, R. Peikert, F. H. Post, and M. Chen, “Over Two Decades of Integration-Based, Geometric Flow Visualization,” *Computer Graphics Forum*, 2010.

[113] E. Hairer, S. P. Nørsett, and G. Wanner, *Solving Ordinary Differential Equations I (2Nd Revised. Ed.): Nonstiff Problems.* New York, NY, USA: Springer-Verlag New York, Inc., 1993.

[114] D. Sujudi and R. Haimes, “- integration of particle paths and streamlines in a spatially-decomposed computation,” in *Parallel Computational Fluid Dynamics 1995* (A. Ecer, J. Periaux, N. Satdfuka, and S. Taylor, eds.), pp. 315 – 322, Amsterdam: North-Holland, 1996.

[115] 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*, pp. 57–64, Oct 2011.

[116] 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)*, pp. 1–6, Oct 2013.

[117] 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*, pp. 580–591, May 2011.

[118] L. Chen and I. Fujishiro, “Optimizing parallel performance of streamline visualization for large distributed flow datasets,” in *2008 IEEE Pacific Visualization Symposium*, pp. 87–94, March 2008.

[119] H. Yu, C. Wang, and K. Ma, “Parallel hierarchical visualization of large time-varying 3d vector fields,” in *SC ’07: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, pp. 1–12, Nov 2007.

[120] 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*, vol. 17, pp. 1785–1794, Dec 2011.

[121] 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*, SC '09, (New York, NY, USA), pp. 53:1–53:11, ACM, 2009.

[122] P. J. Rhodes, X. Tang, R. D. Bergeron, and T. M. Sparr, "Iteration aware prefetching for large multidimensional datasets," in *Proceedings of the 17th International Conference on Scientific and Statistical Database Management*, SSDBM'2005, (Berkeley, CA, US), pp. 45–54, Lawrence Berkeley Laboratory, 2005.

[123] O. O. Akande and P. J. Rhodes, "Iteration aware prefetching for unstructured grids," in *2013 IEEE International Conference on Big Data*, pp. 219–227, Oct 2013.

[124] H. Childs, D. Pugmire, S. Ahern, B. Whitlock, M. Howison, Prabhat, G. H. Weber, and E. W. Bethel, "Extreme scaling of production visualization software on diverse architectures," *IEEE Computer Graphics and Applications*, vol. 30, pp. 22–31, May 2010.

[125] 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, (New York, NY, USA), pp. 10:1–10:11, ACM, 2011.

[126] H. Guo, X. Yuan, J. Huang, and X. Zhu, "Coupled ensemble flow line advection and analysis," *IEEE Transactions on Visualization and Computer Graphics*, vol. 19, pp. 2733–2742, Dec. 2013.

[127] 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*, pp. 33–40, March 2014.

[128] 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)*, pp. 96–103, April 2016.

[129] K. Lu, H. 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*, pp. 1008–1019, Nov 2014.

[130] D. Camp, H. Childs, C. Garth, D. Pugmire, and K. I. Joy, "Parallel stream surface computation for large data sets," in *IEEE Symposium on Large Data Analysis and Visualization (LDAV)*, pp. 39–47, Oct 2012.

[131] K. Lu, H. 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*, pp. 1008–1019, Nov 2014.

[132] P. F. Fischer, J. W. Lottes, and S. G. Kerkemeier, “nek5000 Web page,” 2008. http://nek5000.mcs.anl.gov.

[133] 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,” *J. Comp. Phys.*, vol. 195, p. 355, 2004.

[134] E. Endeve, C. Y. Cardall, R. D. Budiardja, and A. Mezzacappa, “Generation of Magnetic Fields By the Stationary Accretion Shock Instability,” *The Astrophysical Journal*, vol. 713, no. 2, pp. 1219–1243, 2010.

[135] R. Sisneros and D. Pugmire, “Tuned to terrible: A study of parallel particle advection state of the practice,” in *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 1058–1067, May 2016.

[136] Intel Corporation, “Introducing the Intel Threading Building Blocks,” May 2017. `https://software.intel.com/en-us/node/506042`.

[137] “Mpi: A message passing interface,” in *Supercomputing ’93. Proceedings*, pp. 878–883, Nov 1993.

[138] Vetter and Olbrich, “Development and integration of an in-situ framework for flow visualization of large-scale, unsteady phenomena in icon,” *Supercomput. Front. Innov.: Int. J.*, vol. 4, pp. 55–67, Sept. 2017.

[139] A. Agranovsky, D. Camp, C. Garth, E. W. Bethel, K. I. Joy, and H. Childs, “Improved Post Hoc Flow Analysis via Lagrangian Representations,” in *Proceedings of the IEEE Symposium on Large Data Visualization and Analysis (LDAV)*, pp. 67–75, Nov. 2014.

[140] S. Sane, R. Bujack, and H. Childs, “Revisiting the Evaluation of In Situ Lagrangian Analysis,” in *Eurographics Symposium on Parallel Graphics and Visualization (EGPGV)*, (Brno, Czech Republic), pp. 63–67, June 2018.

[141] S. Sane, H. Childs, and R. Bujack, “An Interpolation Scheme for VDVP Lagrangian Basis Flows,” in *Eurographics Symposium on Parallel Graphics and Visualization (EGPGV)*, (Porto, Portugal), pp. 109–118, June 2019.

[142] P. Balaji, R. Gupta, A. Vishnu, and P. Beckman, “Mapping communication layouts to network hardware characteristics on massive-scale blue gene systems,” *Comput. Sci.*, vol. 26, p. 247–256, June 2011.

[143] T. Fujiwara, P. Malakar, K. Reda, V. Vishwanath, M. E. Papka, and K. Ma, “A visual analytics system for optimizing communications in massively parallel applications,” in *2017 IEEE Conference on Visual Analytics Science and Technology (VAST)*, pp. 59–70, Oct 2017.

[144] M. Larsen, J. Ahrens, U. Ayachit, E. Brugger, H. Childs, B. Geveci, and C. Harrison, “The alpine in situ infrastructure: Ascending from the ashes of strawman,” in *Proceedings of the In Situ Infrastructures on Enabling Extreme-Scale Analysis and Visualization*, ISAV’17, (New York, NY, USA), pp. 42–46, ACM, 2017.