IN-LINE VS. IN-TRANSIT IN SITU: WHICH TECHNIQUE TO USE AT SCALE?

by

JAMES MICHAEL KRESS

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: James Michael Kress

Title: In-line vs. In-transit In Situ: Which Technique to Use at Scale?

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
Allen Malony	Core Member
Boyana Norris	Core Member
Victor Ostrik	Institutional Representative
David Pugmire	Outside Member

and

Kate Mondloch	Interim 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

DISSERTATION ABSTRACT

James Michael Kress

Doctor of Philosophy

Department of Computer and Information Science

March 2020

Title: In-line vs. In-transit In Situ: Which Technique to Use at Scale?

In situ visualization is increasingly necessary to address I/O limitations on supercomputers. With the increasing heterogeneity of supercomputer design, efficient and cost effective use of resources is extremely difficult for in situ visualization routines. In this work, we present a time and cost analysis of two different classes of common visualization algorithms in order to determine which in situ paradigm (in-line or in-transit) to use at scale, and under what circumstances. We explore a high computation and low communication algorithm, as well as a low computation and medium communication algorithm. We use 255 individual experimental runs to compare these algorithms performance at scale (up to 32,768 cores in-line and 16,384 core in-transit) with a running simulation. Finally, we show that — contrary to community belief — in-transit in situ has the potential to be both faster and more cost efficient than in-line in situ. We term this discovery *Visualization Cost Efficiency Factor (VCEF)*, which is a measure of how much more performant in-transit in situ is on a smaller subset of nodes than in-line in situ is at the full scale of a simulation. Our results for these algorithms showed in-transit *VCEF* values of up to *8X* at our highest concurrencies.

This dissertation includes previously published co-authored material.

CURRICULUM VITAE

NAME OF AUTHOR: James Michael Kress

GRADUATE AND UNDERGRADUATE SCHOOLS ATTENDED:

University of Oregon, Eugene, OR, USA
Boise State University, Boise, ID, USA

DEGREES AWARDED:

Doctor of Philosophy, Computer and Information Science, 2020, University of Oregon
Master of Science, Computer and Information Science, 2016, University of Oregon
Bachelor of Science Computer Science, 2013, Boise State University, Cum Laude. Minor in Political Science

AREAS OF SPECIAL INTEREST:

In Situ Visualization
Scientific Visualization
High Performance Computing

PROFESSIONAL EXPERIENCE:

Staff Scientist, Oak Ridge National Laboratory, 2016–Present
Graduate Research Assistant, University of Oregon, 2013-2016
Student Research Intern, Oak Ridge National Laboratory, Summer 2016
Student Research Intern, Oak Ridge National Laboratory, Summer 2015
Student Research Intern, Oak Ridge National Laboratory, Summer 2014
Student Research Intern, Oak Ridge National Laboratory, Summer 2013

GRANTS, AWARDS AND HONORS:

Area Exam, Passed With Distinction, University of Oregon, 2017

PUBLICATIONS:

J. Kress, M. Larsen, J. Choi, M. Kim, M. Wolf, N. Podhorszki, S. Klasky, H. Childs, and D. Pugmire. "Analyzing the Time Savings of In Situ Visualization Paradigms at Scale" (In Preparation)

J. Kress, M. Larsen, J. Choi, M. Kim, M. Wolf, N. Podhorszki, S. Klasky, H. Childs, and D. Pugmire. "Opportunities for Cost Savings with In-transit Visualization" In: ISC High Performance 2020. (Accepted for Publication)

W. F. Godoy, N. Podhorszki, R. Wang, C. Atkins, G. Eisenhauer, J. Guo, P. Davis, J. Choi, K. Germaschewski, K. Huck, A. Huebl, M. Kim, **J. Kress**, T. Kurc, Q. Liu, J. Logan, K. Mehta, G. Ostrouchov, M. Parashar, F. Poeschel, D. Pugmire, E. Suchyta, K. Takahashi, N. Thompson, S. Tsutsumi, L. Wan, M. Wolf, J. Wu, and S. Klasky, ADIOS2: The Adaptable Input Output System. A Framework for High-Performance Data Management, Software-X 2020. (Accepted for Publication)

J. Kress, M. Larsen, J. Choi, M. Kim, M. Wolf, N. Podhorszki, S. Klasky, H. Childs, and D. Pugmire. "Comparing the Efficiency of In Situ Visualization Paradigms at Scale." In: ISC High Performance 2019.

J. Kress, J. Choi, S. Klasky, M. Churchill, H. Childs, and D. Pugmire. "Binning Based Data Reduction for Vector Field Data of a Particle-In-Cell Fusion Simulation." In: ISC High Performance 2018 International Workshops. Vol. 11203. Lecture Notes in Computer Science. Frankfurt, Germany: Springer Publishing, June 2018, pp. 215–229.

M. Kim, **J. Kress**, J. Choi, N. Podhorszki, S. Klasky, M. Wolf, K. Mehta, K. Huck, B. Geveci, S. Phillip, et al. "In Situ Analysis and Visualization of Fusion Simulations: Lessons Learned." In: ISC High Performance 2018 International Workshops. Vol. 11203. Lecture Notes in Computer Science. Frankfurt, Germany: Springer Publishing, June 2018, pp. 230–242.

J. Choi, CS Chang, J. Dominski, S. Klasky, G. Merlo, E. Suchyta, M. Ainsworth, B. Allen, F. Cappello, M. Churchill, P. Davis, S. Di, G. Eisenhauer, S. Ethier, I. Foster, B. Geveci, H. Guo, K. Huck, F. Jenko, M. Kim, **J. Kress**, S. Ku, Q. Liu, J. Logan, A. Malony, K. Mehta, K. Moreland, T. Munson, M. Parashar, T. Peterka, N. Podhorszki, D. Pugmire, O. Tugluk, R. Wang, B. Whitney, M. Wolf, and C. Wood. "Coupling Exascale Multiphysics Applications: Methods and Lessons Learned." In: 2018 IEEE 14th International Conference on e-Science (e-Science) (2018), pp. 442–452.

D. Pugmire, A. Yenpure, M. Kim, **J. Kress**, R. Maynard, H. Childs, and B. Hentschel. "Performance-Portable Particle Advection with VTK-m." In: EGPGV. 2018, pp. 45–55.

S. Klasky, M. Wolf, M. Ainsworth, C. Atkins, J. Choi, G. Eisenhauer, B. Geveci, W. Godoy, M. Kim, **J. Kress**, T. Kurc, Q. Liu, J. Logan, A. B. Maccabe, K. Mehta, G. Ostrouchov, M. Parashar, N. Podhorszki, D. Pugmire, E. Suchyta, L. Wan, and R. Wang. "A View from ORNL: Scientific Data Research Opportunities in the Big Data Age." In: 2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS). IEEE. 2018, pp. 1357–1368.

J. Kress, D. Pugmire, S. Klasky, H. Childs. "Visualization and Analysis Requirements for In Situ Processing for a Large-scale Fusion Simulation Code." Proceedings of the 2nd Workshop on In Situ Infrastructures for Enabling Extreme-scale Analysis and Visualization. IEEE Press, 2016.

M. Larsen, C. Harrison, **J. Kress**, D. Pugmire, J. Meredith, and H. Childs. "Performance Modeling of In Situ Rendering." High Performance Computing, Networking, Storage and Analysis, SC16. IEEE, 2016.

J. Kress, R. M. Churchill, S. Klasky, M. Kim, H. Childs, and D. Pugmire. "Preparing for In Situ Processing on Upcoming Leading-edge Supercomputers." Supercomputing frontiers and innovations 3.4 (2016): 49-65.

D. Pugmire, **J. Kress**, J. Choi, S. Klasky, T. Kurc, R.M. Churchill, M. Wolf, G. Eisenhower, H. Childs, K. Wu, and A. Sim. "Visualization and Analysis for Near-real-time Decision Making in Distributed Workflows." Parallel and Distributed Processing Symposium Workshops, 2016 IEEE International. IEEE, 2016.

K. Moreland, C. Sewell, W. Usher, L.T. Lo, J. Meredith, D. Pugmire, **J. Kress**, H. Schroots, K. Ma, H. Childs, and M. Larsen. "VTK-m: Accelerating the Visualization Toolkit for Massively Threaded Architectures." IEEE computer graphics and applications 36.3 (2016): 48-58.

J. Kress, S. Klasky, N. Podhorszki, J. Choi, H. Childs, and D. Pugmire. "Loosely Coupled In Situ Visualization: A Perspective on Why It's Here to Stay." Proceedings of the First Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization. ACM, 2015.

J. Kress, E. Anderson, and Hank Childs. "A Visualization Pipeline for Large-scale Tractography Data." Large Data Analysis and Visualization (LDAV), 2015 IEEE 5th Symposium on. IEEE, 2015.

D. Pugmire, **J. Kress**, J. Meredith, N. Podhorszki, J. Choi, and S. Klasky. "Towards Scalable Visualization Plugins for Data Staging Workflows." Big Data Analytics: Challenges and Opportunities (BDAC-14) Workshop at Supercomputing Conference. 2014

J. Kress, S. Xu, and G. Tourassi. "A Novel Graphical User Interface for High-efficacy Modeling of Human Perceptual Similarity Opinions." Medical Imaging 2013: Image Perception, Observer Performance, and Technology Assessment. Vol. 8673. International Society for Optics and Photonics, 20133

ACKNOWLEDGEMENTS

First, I would like to thank Dr. Hank Childs for his unique and dedicated advising style. You were a light in the dark on this exploration, and encouraged me to pursue my interests, always there with helpful guidance and direction. Furthermore, your technical and professional expertise has made me into a much better writer and a more confident researcher. You made the years of grad school not only great, but invaluable.

To my undergraduate advisor Dr. Alark Joshi, thank you for enlightening me to education beyond my undergraduate degree. Your insight, encouragement, and friendship connected me with my graduate advisor, and have led to a whole new world of exploration.

I would like to thank my committee members, Dr. Allen Malony, Dr. Boyana Norris, Dr. Victor Ostrik, Dr. David Pugmire, and Dr. Hank Childs. Your feedback and discussions during this process made for more successful and technically sound research and results.

I would also like to thank my Oak Ridge National Laboratory advisor, colleague, and friend, Dr. David Pugmire. This research would not have been possible without you and your vision of the future of in situ visualization. Your advisement, encouragement, and expertise led me to see in situ visualization differently, opening up many interesting areas of future research exploration.

Thank you to my CDUX cohort, especially Ryan Bleile, Stephanie Brink (Labasan), Matthew Larsen, and Shaomeng Li. You made the entire time at UO a great experience, from the late night paper editing sessions to TV watch parties. Your technical expertise was invaluable for coding questions, reviewing early

iterations of papers and research ideas, all the while doing much of that remotely as we all moved off to our various new institutions.

Some of this research was funded in part by the DOE Early Career Award, Contract No. DE-SC0010652, Program Manager Lucy Nowell. Some of this work was funded by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Scientific Discovery through Advanced Computing (SciDAC) program. Some of this research was performed by UT-Battelle, LLC, with the US Department of Energy. Some of this work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344. This research used resources of the Oak Ridge Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC05-00OR22725.

I dedicate this dissertation to my parents. You taught me the value of hard work, dedication, and what it takes to accomplish a goal. Your unwavering support and encouragement is what allowed me to pursue my deams. Thank you to my wife for your understanding, support, encouragement, and love throughout this process. Our adventures together always offer me clarity. Without all of you, this achievement would not have been possible.

TABLE OF CONTENTS

Chapter Page

Chapter Page

LIST OF FIGURES

LIST OF TABLES

CHAPTER I

INTRODUCTION

1.1 Motivation

As leading-edge supercomputers get increasingly powerful, scientific simulations running on these machines are generating ever larger volumes of data. However, the increasing cost of data movement, in particular moving data to disk, is increasingly limiting the ability to process, analyze, and fully comprehend simulation results [12], hampering knowledge extraction. Specifically, while I/O bandwidths regularly increase with each new supercomputer, these increases are well below corresponding increases in computational ability and data generated. Further, this trend is predicted to persist for the foreseeable future.

Traditionally, visualization has been performed as a post-processing task, where simulation outputs are read from disk into the memory of a parallel tool which performs analysis and visualization tasks. Visualization is generally I/O bound [39, 40], and as relative I/O bandwidth continues to decrease, the challenges of visualizing increasingly larger data will become more problematic. In the case of traditional visualization, the I/O bottleneck is exacerbated as data is first written to disk by the simulation, and then read back from disk by the visualization routine.

Largely due to the increasing I/O bottleneck, in-situ analysis and visualization techniques are receiving significant attention. These techniques operate on simulation data *as* they are produced, as opposed to *after* they are produced, which is the traditional use case for post-processing analysis and visualization of data on disk. In addition to alleviating the I/O bottleneck, these techniques have the added benefit of access to *all* of the simulation data, and

since simulations typically only output a limited set of time steps to disk, these techniques have access to every time step.

Broadly speaking, two paradigms have emerged [38]. The first paradigm is *co-processing*, or *in-line* methods. In this dissertation, we define in-line to mean when the simulation and visualization code run in the same process using the same resources. The second paradigm is *concurrent-processing*, or *in-transit* methods. In this dissertation, we define in-transit to mean when the simulation transfers data over the network to a separate set of visualization nodes for processing. For simplification, we view these two paradigms as on-node and off-node. In-line can be thought of as running on the same node as the simulation, and not utilizing asynchronous data transfers from the simulation to the visualization routines, while in-transit can be viewed as off-node.

In a 2015 position paper [70], I proposed a set of 10 comparison factors that enable concrete comparisons to be made based on the costs and benefits associated with each of these in situ scenarios. These factors consider required HPC resources (both shared and dedicated), impact on the running simulation, fault tolerance, and usability. These factors are discussed in depth in Chapter IV, and a high level overview of these factors can be found in Table 1. The outcome of this position paper was a set of opinions on which in situ paradigm benefited the most from each of the 10 comparison factors. Some of these factors are hard to empirically test, and are very situationally dependent. Others however, can be directly tested and assertions about superiority can be proven (or disproven).

Of the ten comparison factors we proposed, the factor with the most immediate impact to end users is *Scalability*. If a visualization algorithm is run on a very large resource at high concurrency, and it does not scale, that run will

Table 1. Overview of the 10 different factors we devised for comparing the benefits of both the in-line and in-transit in situ paradigms. The paradigm which the position paper asserted to be the strongest in a given category is indicated with a check mark, and a dash is used when the paradigms are thought to be equally as good.

		Favored Paradigm	
	Comparison Factor	**In-line**	**In-transit**
Data Factors	Data Access	✓	
	Data Movement	—	—
	Data Duplication	✓	
	Data Translation		✓
	Exploratory Visualization	—	—
Implementation Factors	Scalability		✓
	Ease of Use		✓
	Coordination	✓	
	Fault Tolerance		✓
	Resource Requirements	—	—

incur a heavy penalty. However, if the same visualization algorithm is run at a lower concurrency, it may not incur that same penalty. This is why understanding the scalability of visualization algorithms in the context of in situ is important. It has the potential to save users of visualization algorithms both time and money as they will not have to spend as much supercomputer time performing visualization if they can know ahead of time what concurrency will be the most cost effective.

1.2 Research Goals and Approaches

The central question that this dissertation addresses is: *"In-line vs. in-transit in situ: which technique to use at scale?"* Of the ten comparison factors we proposed for in situ, this dissertation focuses exclusively on the *Scalability* factor, and beyond scalability to overall cost (total compute time over all resources). We

will show through experimentation and modeling which in situ paradigm performs the best and under what circumstances.

This is an important question for the visualization community because the impacts of using different visualization algorithms in-transit or in-line at various different scales and resource allocations is not well understood. There are a variety of things that affect performance, ranging from the size of the visualization allocation used, the scale of the simulation, the frequency of visualization, and the characteristics of the visualization algorithms themselves. This complexity leaves a number of open questions that can be addressed by a scalability study:

- **Q:** How does communication between ranks affect in-line visualization (is it more efficient for some algorithms vs. others)?
- **Q:** What size of resource allocation is needed for in-transit visualization so that resources are not wasted when doing infrequent visualization?
- **Q:** At lower concurrency, are in-line techniques more efficient?
- **Q:** What are the overheads associated with in-transit techniques?
- **Q:** Does in-transit ever cost less to use than in-line?
- **Q:** What percentage of simulation resources are needed for in-transit so that it does not block the simulation (so that it keeps up)?

To answer these sub questions about *scalability* and the overall cost of using each in situ paradigm, we developed an in situ workflow to test two different common visualization algorithms at differing levels of concurrency under each of these two paradigms. We tested these algorithms from low (128 cores) to high (32,768 cores) concurrency to determine how each algorithm performed at different scales under both in situ paradigms. It was critical to test a broad range

of concurrencies for this study due to the changing behavior of different algorithms at different concurrencies. The two visualization operations that were selected were picked because they cover the gamut of important visualization operations to the visualization community. When the VTK-m [103] project was creating their proposal for funding under the Exascale Computing Project (under which they are now funded), they developed a list of algorithms that were critical to be implemented during the first phase of the project, as well as some aspirational algorithms for future development. That list was:

1. Point Location
2. Cell Location
3. Clipping
4. Point Merging
5. Connected Components
6. Particle Advection
7. Particle Advection Time Varying (Pathlines)
8. Contouring
9. External Surfaces
10. Ray Tracing
11. Volume Rendering
12. Particle/Glyph Rendering

The two algorithms we selected for evaluation were Volume Rendering, and Isosurfacing. These two algorithms each fall into a different class of algorithms where the amount of work and communication between parallel process differs highly. Isosurfacing is computationally bound, and does little communication, whereas volume rendering does much more communication, changing it from a computation-bound to a communication-bound algorithm. The differences in the core characteristics of these algorithms make them good candidates for evaluating the question of scalability and cost effectiveness at scale.

1.3 Dissertation Outline

Putting it all together, choosing the correct configuration for in situ visualization is challenging, as the most cost effective solution may vary from run to run, depending on a host of factors. There are currently very few works that address the challenges of choosing a cost efficient configuration for in situ, and none that explore multiple visualization algorithms at scale. The goal of this work is to explore this space in order to help others that want to both utilize the benefits of in situ visualization, but also wish to use their limited compute resources to their fullest potential.

The remainder of this dissertation is organized into two parts as follows:

- Part I: Foundations
  - Chapter II: We survey past works in high performance computing, visualization, and in situ visualization. This survey provides a foundation for what the current state of the art is in visualization, and provides a point of reference for our developments and findings in Part II of the dissertation.
  - Chapter III: We survey the members of a large-scale fusion simulation code in order to gather their requirements for visualization and analysis. We look at these requirements from the perspective of in situ processing, and present of a list of their needs for current and future visualization and analysis.
  - Chapter IV: We develop a set of 10 factors for comparing in-line and in-transit in situ techniques, including the factor that this dissertation is centered around.

– Part II: Findings

  * Chapter V: We present the setup, configurations, and preliminary results from 255 individual in situ visualization test case runs on the Titan supercomputer. This is our corpus of data that we analyze in the subsequent two chapters.

  * Chapter VI: We evaluate our corpus of data from the perspective that time-is-of-the-essence for an in situ visualization task, and discuss the primary factors effecting which in situ paradigm is the fastest as application concurrency increases.

  * Chapter VII: We evaluate our corpus of data from the perspective that total cost to the user is the primary driver for in situ visualization. We then develop a model for predicting the cost efficiency of in-line and in-transit visualization configurations.

  * Chapter VIII: We conclude by summarizing our key findings and discoveries, and take a significant look at directions for the most interesting future work.

1.4 Co-Authored Material

Much of the work in this dissertation is from previously published co-authored material. Below is a listing connecting the chapters with the publications and authors that contributed. Additional detail on the division of labor for each publication is provided at the beginning of each chapter. That said, for each of these publications, I was not only the first-author of the paper, but also the primary contributor for implementing systems, conducting studies, and writing manuscripts.

– Chapter II: This chapter is composed of portions of my Ph.D. Area Exam document which was unpublished.

– Chapter III: [74] was a collaboration between Scott Klasky (ORNL), David Pugmire (ORNL), Hank Childs (UO, LBL), and myself.

– Chapter IV: [70] was a collaboration between Scott Klasky (ORNL), Norbert Podhorszki (ORNL), Jong Choi (ORNL), Hank Childs (UO and LBL), David Pugmire (ORNL) and myself.

– Chapter V: This chapter summarizes the data that was gathered and analyzed in the following two chapters, so is composed of components of two different works, [71, 73], which were collaborations between Matthew Larsen (LLNL), Jong Choi (ORNL), Mark Kim (ORNL), Matthew Wolf (ORNL), Norbert Podhorszki (ORNL), Scott Klasky (ORNL), Hank Childs (UO), David Pugmire (ORNL), and myself.

– Chapter VI: [71] was a collaboration between Matthew Larsen (LLNL), Jong Choi (ORNL), Mark Kim (ORNL), Matthew Wolf (ORNL), Norbert Podhorszki (ORNL), Scott Klasky (ORNL), Hank Childs (UO), David Pugmire (ORNL), and myself.

– Chapter VII: [73] was a collaboration between Matthew Larsen (LLNL), Jong Choi (ORNL), Mark Kim (ORNL), Matthew Wolf (ORNL), Norbert Podhorszki (ORNL), Scott Klasky (ORNL), Hank Childs (UO), David Pugmire (ORNL), and myself.

Part I

Foundations

In this part of the dissertation, we provide background on in situ visualization techniques (Chapter II), survey a large simulation code to gather visualization requirements to motivate in situ (Chapter III), and conclude with a set of factors for evaluating in situ techniques (Chapter IV). These chapters provide the foundations on which the rest of this dissertation is based.

CHAPTER II

BACKGROUND

As leading-edge supercomputers get increasingly powerful, scientific simulations running on these machines are generating ever larger volumes of data. However, the increasing cost of data movement, in particular moving data to disk, is increasingly limiting the ability to process, analyze, and fully comprehend simulation results [12], hampering knowledge extraction. Specifically, while I/O bandwidths regularly increase with each new supercomputer, these increases are well below corresponding increases in computational ability and data generated. Further, this trend is predicted to persist for the foreseeable future.

Relative decreases in I/O pose a problem for stakeholders running on these systems ranging from simulation scientists to visualization researchers. To that end, the Advanced Scientific Computing Research (ASCR) Scientific Grand Challenges Workshop Series produced reports spanning eight different scientific domains (High Energy Physics, Climate, Nuclear Physics, Fusion, Nuclear Energy, Basic Energy Sciences, Biology, National Security) [28, 138, 144, 128, 119, 54, 126, 27], that explored the computing challenges, including visualization and analysis challenges, for codes in each of those eight domains. Each report mentioned data movement, storage, and analysis as a major obstacle in the move to exascale. Many of these scientific domains will be required to deal with petabytes, or even exabytes, of data over the course of a simulation.

This trend poses a problem for the traditional post-processing visualization methodology. The traditional visualization workflow performs visualization as a post-processing task, where simulation outputs are read from disk, into the memory of a parallel tool which performs analysis and visualization. Visualization is

generally I/O bound [39, 40], and as relative I/O bandwidth continues to decrease, the challenges of visualizing increasingly larger data will become more problematic. Post hoc visualization is particularly sensitive to the I/O bottleneck, as data is first written to disk by the simulation, and then read back from disk by the visualization routine.

Given this reality, many large-scale simulation codes are attempting to bypass the I/O bottleneck by using in situ visualization and analysis, i.e., processing simulation data when it is generated. Broadly speaking, two paradigms have emerged [38]. First, *co-processing*, or *in-line*, methods, where the simulation and visualization code run in the same process using the same resources. Second, *concurrent-processing*, or *in-transit*, methods, where the simulation transfers data over the network to a separate set of visualization nodes for processing.

In situ processing poses many new challenges to both simulation and visualization scientists that were hidden or less predominant with the post-processing paradigm. A few of the issues facing in situ include: how the in situ routines are integrated with the simulation, how data is translated from the simulation representation to the visualization representation, how resources are allocated between the simulation and the visualization, how faults are isolated in the visualization routines, how to massively scale communication heavy visualization algorithms, and even how to do exploratory visualization in an in situ world. One avenue of approach that specifically address the resource allocation and scalability problems is the modeling of visualization algorithms under varying computational setups and data loads. This modeling work is an exciting area of future research for in situ.

In the remainder of this Chapter, we survey and explore in situ visualization itself, key areas involved in in situ workflows, and identify areas where the research is incomplete, or requires further study. First, we look at trends in high performance computing and their implications for the future of visualization in Section 2.1. Next, we explore the traditional scientific visualization and compositing pipelines, and discuss prevalent scientific visualization tools including current research in the area of data models, portable performance, and massive scale visualization in Section 2.2. And finally, the state of in situ visualization is discussed in Section 2.3.

2.1 High Performance Computing

High Performance Computing (HPC) is a landscape of constant evolution. This evolution is seen in the composition of the HPC systems themselves, as well as the science that they enable. By using these systems, scientists have gained deeper understandings in fields ranging from medicine to energy to physics to even national security. Computers have seen nearly a 50 billion-fold increase in computing power over the last 70 years [108]. Compared to other technologies, this is virtually an unprecedented leap, enabling more than ever before, but bringing with it a vast set of challenges.

One of those primary challenges is power. The Department of Energy has set a nominal power cap for exascale systems at 20 MW per year. This roughly equates to a yearly energy bill of $20 million dollars. However, reaching this goal is not easy. It would be possible to construct an exascale system today using conventional hardware and components, but DARPA estimated in 2008 that this system's power requirements would reach into the 100's of MW, far beyond the

Table 2. Previous, current, and next generation system statistics for Advanced Scientific Computing Research Programs computing resources. Two areas of critical importance to note are the node processors and the system size of the previous machines compared to the current evolution. Visualization codes are expected to work efficiently on concurrencies and architectures never seen before, meaning that the challenges from exascale computing are already emerging now (modified table from [59]).

System attributes	NERSC Prior	OLCF Prior	ALCF Prior	NERSC Upgrade	OLCF Upgrade	ALCF Upgrades	
Name Planned Installation	**Edison**	**TITAN**	**MIRA**	**Cori 2016**	**Summit 2019**	**Theta 2016**	**Aurora 2021**
System peak (PF)	2.6	27	10	> 30	150	>8.5	180
Peak Power (MW)	2	9	4.8	< 3.7	10	1.7	13
Total system memory	357 TB	710TB	768TB	~1 PB DDR4 + High Bandwidth Memory (HBM)+1.5PB persistent memory	> 1.74 PB DDR4 + HBM + 2.8 PB persistent memory	>480 TB DDR4 + High Bandwidth Memory (HBM)	> 7 PB High Bandwidth On - Package Memory Local Memory and Persistent Memory
Node performance (TF)	0.460	1.452	0.204	> 3	> 40	> 3	> 17 times Mira
Node processors	Intel Ivy Bridge	AMD Opteron Nvidia Kepler	64-bit PowerPC A2	Intel Knights Landing many core CPUs Intel Haswell CPU in data partition	Multiple IBM Power9 CPUs & multiple Nvidia Voltas GPUS	Intel Knights Landing Xeon Phi many core CPUs	Knights Hill Xeon Phi many core CPUs
System size (nodes)	5,600 nodes	18,688 nodes	49,152	9,300 nodes 1,900 nodes in data partition	~3,500 nodes	>2,500 nodes	>50,000 nodes
System Interconnect	Aries	Gemini	5D Torus	Aries	Dual Rail EDR-IB	Aries	2nd Generation Intel Omni-Path Architecture
File System	7.6 PB 168 GB/s, Lustre®	32 PB 1 TB/s, Lustre®	26 PB 300 GB/s GPFS™	28 PB 744 GB/s Lustre®	120 PB 1 TB/s GPFS™	10PB, 210 GB/s Lustre initial	150 PB 1 TB/s Lustre®

maximum power bound [23]. This estimate has since dropped with new system designs being introduced, but it is still far beyond the 20 MW cap.

Therefore, to reach the performance goal given the maximum power bound, system designers are having to divert from the traditional approach for scaling HPC systems, by transitioning them from multi-core to many-core. This transition pares down the power of the traditional central processing unit in each node of the supercomputer, and instead, gets its performance by utilizing many low power cores on devices such as GPUs and Intel Xeon Phis. Indeed, this trend is already being

seen as the current generation of Department of Energy computing systems are being prepared for their 2018 redesigns/upgrades.

Table 2 shows the three DOE supercomputing systems that underwent upgrades in the 2016 to 2018 time frame. Focusing on just Titan, a drastic change took place in the topology of this system when it was replaced by Summit (the current fastest computer in the world [9]). Titan contained 18,688 nodes, consumes a total of 9 MW of power, and had a peak performance of 27 PF. However, Summit drastically cut the number of nodes in the system down to just around 3,500 nodes, a total power consumption of 10 MW, and a peak performance of 150 PF. This change highlights that the challenges of exascale are already here. Moving from a system that had has million-way concurrency to a system with billion-way concurrency necessitates a redesign of not only the simulations and codes running on this system (focusing on parallelizing the underlying algorithms) [117], but also in how data is saved and analyzed [17].

Taking it one step further, Table 3 shows the expected characteristics of an actual machine at exascale. This table focuses on the system performance versus the system I/O, in order to highlight the data challenge. The system peak performance is expected to rise by a factor of 500, yet the I/O capacity is only expected to rise by a factor of 20. This means that the current problems faced by simulation codes in terms of how frequently they can save data are only going to get worse. Take, for example, the leading-edge fusion simulation code XGC1 which saves time steps on average every 100 steps [74]. Moving this code to an exascale system without addressing the data problem is going to mean that time steps will now only be saved every 1,000 to 10,000 steps. This will drastically increase the likelihood that interesting physics will be lost between saves.

Table 3. Current petascale system performance compared against the design target for the 2023 exascale system. Moving to billion way concurrency and an exaflop in performance are critical challenges for visualization when compared to current visualization algorithm scaling and the network bandwidth when trying to move data to disk (adapted from [99]).

System Parameter	2011	"2023"		Factor Change
System Peak	2 PF	1 EF		500
Power	6 MW	<= 20 MW		3
System Memory	0.3 PB	32 PB	64 PB	100-200
Total Concurrency	225K	1Bx10		40,000
Node Performance	125 GF	1 TF	10 TF	8-80
Node Concurrency	12	1,000	10,000	83-830
Network BW	1.5 GB/s	100 GB/s	1000 GB/s	66-660
System Size (nodes)	18,700	100,000	1,000,000	50-500
I/O Capacity	15 PB	300 PB	1000 PB	20-67

In situ processing can address this with faster analysis of data streams without having to first send data to disk. This means that a higher temporal fidelity of data will be available for analysis, while even potentially enabling the possibility of interactive steering of the simulation through the visualization [14].

2.2 Scientific Visualization

Visualization is an enabling technology that facilitates insight into data across many domains. It is an essential tool for confirming and communicating trends in data to both the domain scientists as well as the general public [37]. Traditionally, scientific visualization has been performed as a post processing task, where a simulation will save all of the data needed for visualization to disk, and after the run is complete, visualization can begin. This approach has the benefit

that the visualization software has access to all of the data from every step all at once, making algorithms and visualization workflows easier to develop.

Most of the parallelism in current scientific visualization tools relies on not just distributed memory parallelism, but specifically the message passing interface (MPI). MPI is heavyweight, and requires a whole copy of the visualization program per process. As we transition our visualization codes to higher and higher concurrencies on the march to exascale, this overhead can exceed the system memory and disk space before any data is even loaded [99]. This revelation is important to consider when running a visualization tool at scales approaching those the size of the scientific simulations themselves.

In order to achieve parallel scalability for massive threading, visualization algorithms will have to be redesigned [100]. The key in this redesign will be to focus on data model, data interdependencies, and portable performance. In the following subsections, we will focus on two specific themes in visualization:

1. Section 2.2.1 will look at tools currently being used and developed by the visualization community in terms of their scalability, data models, and challenges for exascale.

2. Section 2.2.2 will explore current trends in graphics for visualization, focusing on image generation in a highly parallel environment.

2.2.1 High Profile Scientific Visualization Tools. There are several tools for scientific visualization that have gained wide adoption and use in the community. Central to the performance of each of these tools are their underlying data models and implementations. In the following sections we will describe several high profile tools for scientific visualization, including how they

handle data on a high level, describing how this impacts performance and use on future systems, and implications for in situ visualization.

2.2.1.1 AVS and OpenDX. The Application Visualization System (AVS) [134] and OpenDX [131] are two early versions of open source visualization tools. AVS is a system that provides a modular interactive approach to forming visualization pipelines. Visualization components are construted visually into flow graphs to create the final visualization product. OpenDX emerged a few years after AVS, and was the open source version of IBM's Data Explorer. OpenDX also had a visual programming interface for constructing visualization pipelines, and contrained many built-in visualization options. These tools have lost prominance with the emergence of newer tools with more refined API's that allow easier integration into existing scientific workflows and batch scheduling systems.

2.2.1.2 VTK. VTK, also known as the Visualization Toolkit [121], is an ongoing software effort enabling extensible visualization and analysis for a wide variety of data set types and filters. The underlying design goals of this toolkit are to be portable, standards based, freely available, and simple [122]. Further, two scalable visualization tools, ParaView [13] and VisIt [36], make use of VTK's foundational data models.

The following discussion of VTK will focus on its data model, as data models are one of the most foundational elements of a visualization tool, and have wide implications in terms of the expressiveness of the data model and its memory overhead in a visualization pipeline.

VTK Data Model VTK's data model exposes a few core mesh types, which are extensible and can be applied to a wide range of scientific domains. The main mesh types supported by VTK are rectilinear, structured, and unstructured. These

three mesh types represent the geometric structure of the data set. Each mesh type consists of point locations in three-dimensional space, cells that reference the area between those points, and fields defined on the points or cells. The fields are stored as values in any number of arrays of data, which can be aligned on the points or cells, or unaligned. The values can range from simple scalar numeric quantities to vector or tensor quantities to more complicated types, such as strings.

VTK Data Model Shortcomings for In Situ and Future Architectures

The main shortcoming of VTK's data model is related to the expressiveness of the model itself in accurately and efficiently representing the multitude types of data produced by simulation codes. VTK's data model supports only a small number of mesh types, such as unstructured and rectilinear grids, but contemporary simulations are representing more complex data instantiations. Even if the data model can accurately represent the simulation data, the data is often forced into an inefficient data structure because VTK has assumed the data will fall into one of the few defined mesh types. Often times, the data does not fit into one of these structures, so it must be forced into a less efficient one.

Another shortcoming of VTK's data model is related to parallelism on future architectures. VTK's data model does not support the recent trends in hardware parallelism resulting from accelerators, such as GPUs. Its data model is also limited in that it does not leverage or support data parallelism.

Lastly, the VTK data model poses challenges when operating on very large data. In the general VTK visualization pipeline, a filter is applied to a data set, and the result is a completely new data set. This means that in general, each filter applied to a VTK data set results in a new data set being created, severely bloating memory. This approach to memory management in a data model does not

scale well for in situ approaches, and will be even more problematic on the next generation of supercomputers.

In summary, VTK's data model lacks support for necessary features, non-Cartesian space, dimensionalities greater than three, and mixed-dimensionality elements in a single data set. As we move to the next generation of architectures and continue evolving scientific simulation codes, there is an increasing demand for an improved and more advanced data model that is extensible and can enable us to represent a wider range of data types. These new representations and memory efficiency are especially important for use in situ, when memory use and an easy translation from simulation data representation to visualization data representation is needed.

2.2.1.3 VisIt and ParaView.

VisIt and ParaView are two open source visualization tools developed, at least in part, through the efforts of U.S. National Laboratories. The history of these tools span many years, and will not be presented here. Instead, the primary design philosophy and major features for end-users will be discussed and then compared to the needs of in situ visualization.

VisIt VisIt is an end-user visualization and analysis tool designed to work on very large and diverse data [35]. Moreover, VisIt was designed for more than just data visualization. It lists five primary use cases that it focuses on [36]:

- Visual Exploration: the creation of images from data.
- Debugging: users can locate hard-to-find problems in their data.
- Quantitative Analysis: users can perform quantitative analysis through the interface to ask very complex questions of their data.

– Comparative Analysis: allows different simulation runs or multiple time steps to be compared.

– Communication: users present their findings to a large audience through movies, images, and plots.

Core to the VisIt design is its extensibility. It allows for new components to be inserted by end-users easily. This extensibility and ease of use makes it a very successful tool, one used across a multitude of scientific domains.

VisIt is designed to work as a distributed system. It has a server that utilizes parallel compute capabilities coupled with the client running as the user interface. In addition, VisIt has capabilities of running in situ with LibSim [75], enabling users to utilize the full feature list of VisIt during in situ instrumentation (the in situ capabilities will be explored further in Section 2.3). VisIt has been shown to scale effectively to tens of thousands of cores, and is widely used by scientists running on some of the largest systems all over the world.

In summary, VisIt is a very powerful visualization tool, that is applicable in a wide variety of use cases. However, two limitations do exist when looking at the use of VisIt in situ: VisIt utilizes VTK under the hood, so the data model issues from VTK come into play. In addition, the visualization library is fairly heavy weight, and can cause problems when performing different types of in situ integrations, potentially making it a sub-optimal approach.

ParaView ParaView is another end-user tool for the visualization of large data. ParaView was designed with the philosophy of being open-source and multi-platform, extensible for different architectures, allowing support for distributed computation, and providing an intuitive user interface.

ParaView was designed as a layered architecture, with three distinct layers [13]. The first is VTK, which provides the data model and underlying algorithms. Second is the parallel extension to VTK to allow for streaming and distributed-memory parallel execution. Third is ParaView itself, predominantly composed of the GUI.

ParaView has been shown to scale well in distributed-memory parallel execution mode, on very large data. In addition, ParaView enables in situ integrations through ParaView Catalyst [50] (the in situ capabilities will be explored further in Section 2.3).

In summary, ParaView is a very powerful visualization tool, that is applicable in a wide variety of use cases. However, two limitations do exist when looking at the use of ParaView in situ: ParaView utilizes VTK under the hood, so the data model issues from VTK come into play, in addition (as with VisIt), the visualization library is fairly heavy weight, and can cause problems when performing different types of in situ integrations, potentially making its a sub-optimal approach.

2.2.1.4 EAVL, Dax, PISTON. EAVL [93, 94], Dax [97], and PISTON [83] are three frameworks developed with a mission to explore methods of transitioning visualization algorithms to the available parallelism of emerging many-core hardware architectures targeted for exascale [123].

- ⋆ EAVL (Extreme-scale Analysis and Visualization Library) was developed to address three primary objectives: update the traditional data model to handle modern simulation codes; investigate the efficiency of I/O, computation and memory on an updated data and execution model; and explore visualization algorithms on next-generation architectures.

The heart of the EAVL approach is the data model. EAVL defines more flexible meshes, and data structures which more efficiently supports the traditional types of data supported by de-facto standards like VTK, but also allows for efficient representations of non-traditional data. Examples of non-traditional data includes graphs, mixed data types (e.g., molecular data, high order field data, unique mesh topologies (e.g., unstructured adaptive mesh refinement and quad-trees)).

EAVL uses a functor concept in the execution model to allow users to write operations that are applied to data. The functor concept in EAVL has been abstracted to allow for execution on either the CPU or GPU, and the execution model manages the movement of data to the particular execution hardware.

⋆ The primary strength of the Dax Toolkit is its exploration of achieving high node-level concurrency, at the levels needed for efficient exascale visualization. This is accomplished through the use of *worklets*, which are functions that implement a given algorithm's behavior on an element of a mesh, or a small local neighborhood. The worklets are constrained to be serial and stateless, which enable concurrent scheduling on an unlimited number of threads.

⋆ PISTON was developed with the goal of facilitating the development of visualization and analysis operators that had highly portable performance. The idea being that there are many different architectures that a visualization algorithm may be run on, and developing and tuning algorithms specific to each architecture is an inefficient and undesirable approach for visualization. To that end, PISTON is built on top of Thrust [21], which provides

implementations of data-parallel primitives in CUDA, OpenMP, and TBB. This approach allows algorithms to be implemented once, and ported to the correct architecture at compile time.

In summary, each of these frameworks provided valuable insight into methods for transitioning visualization pipelines to many-core architectures, and to natively supporting in situ visualization. The best elements from each of these frameworks were used to form the foundation for VTK-m.

2.2.1.5 VTK-m. VTK-m is an effort that has merged the best aspects of three previously described projects, EAVL, Dax and PISTON [103]. The motivator behind VTK-m is to create a high-performance portable visualization library. The portable nature of VTK-m is achieved through its use of data-parallel primitives (DPPs), first described by Blelloch [29]. Data-parallel primitives are designed in a way such that a variety of algorithms can be expressed using a relatively small selection of DPPs, such as map, scan, reduce, and so on. These primitives allow VTK-m to be moved between many different architectures without having to redesign each individual visualization routine. Central to the portable nature of VTK-m is the underlying data model, which is similar to that of EAVL, but with even greater freedom.

The data model in VTK-m was designed to be flexible enough to accommodate the myriad of different data layouts of scientific domains that may use VTK-m, while still providing a clear set of semantics. Furthermore, the data representation must be space efficient and be accessible on the different processor types in use (that is, work on both CPU and GPU). As shown in Figure 1, a VTK-m data set consists of three components: cell sets, coordinate systems, and fields. By allowing arbitrary combinations of coordinate systems, cell sets, and fields,

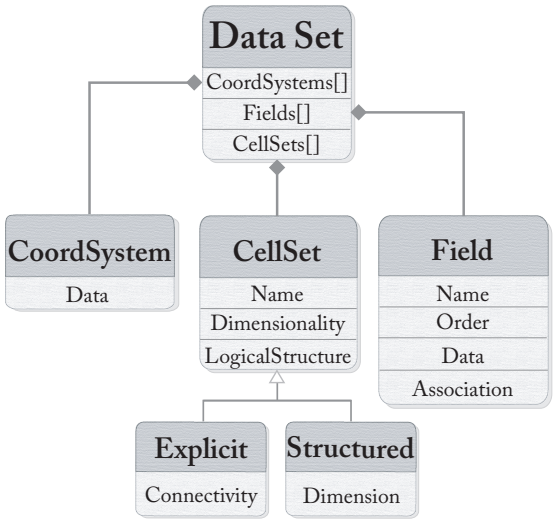


Figure 1. Overview of the VTK-m data model.

VTK-m is able to overcome the inefficiencies and difficulties in data representation imposed by traditional data models. Traditional data models often choose a set of rigid characteristics for a data set. These rigid characteristics then are labeled as a specific type of mesh. For example, a uniform data set has regular axis-aligned coordinates and a logical $[i,\ j,\ k]$ cell arrangement. An unstructured data set has fully explicit coordinates (a $[x,\ y,\ z]$ value separately defined for each point) with fully explicit cell connectivity defined by arrays of indices. This fundamentally rigid way of looking at and representing data makes the traditional data model the less expressive and less efficient choice for high performance computing applications.

VTK-m allows for the much needed more exact representation, and with the burden of the traditional data model removed, VTK-m programmers can create more expressive data layouts. In fact, it is much easier to represent data types such as non-physical or high dimensional data in a VTK-m data model versus that in the traditional paradigm. Another important example of this efficiency is that VTK-m is designed to function with zero copy. This is an important motivator for

in situ programming as VTK-m can utilize the data arrays from the simulation in place, saving both time and space.

In summary, the design directions taken by VTK-m are pushing the current boundaries of visualization from the multi-core realm into the many-core realm, prepping the visualization community for this inevitable transition. VTK-m is being developed as a header only library, which should ease integration issues when using VTK-m in situ, giving it great flexibility.

2.2.2 Graphics in Support of Scientific Visualization. The creation of a graphics system that performs tasks in real-time is a challenging area of study for both graphic system designers as well as scientists employing new graphics algorithms in that space [106]. However, the challenges are justified, as visualization can be one of the most informative methods for communicating the essence of an experiment or data to scientists or the public [49, 114]. With ever increasing geometry and pixel counts, the task of employing an algorithm with a sufficient level of parallelism has become paramount. To that end, there are three basic classes of parallel rendering algorithms recognized in this space, sort-first, sort-middle, and sort-last rending. These algorithms each have been designed for applications in different domains. Sort-last rendering performs best when the geometry is massive compared to the pixel count, commonly seen in HPC visualization. Sort-first on the other hand, is the reverse of sort-last, performing best on low geometry counts with high pixel densities, commonly seen in virtual environment generation. Sort-middle is a hybrid approach that attempts to take the best elements form both sort-first and sort-last.

In this section, we will first describe a basic parallel graphics pipeline and the three techniques for geometry sorting, followed by a discussion of optimized

algorithms for sort-last rendering, and a framework designed to composite images at massive scale. This analysis is important for when we move to discuss in situ visualization, as rendering can be a major bottleneck for in situ visualization tasks.

2.2.2.1 A Parallel Graphics Pipeline. The heart of a parallel graphics pipeline can be viewed as a sorting problem, where the contribution of each object in a given view by each pixel must be determined. The location of this sort determines the entire structure of the resulting parallel algorithm. The sort can, in general, take place anywhere in the rendering pipeline: during geometry processing (sort-first), between geometry processing and rasterization (sort-middle), or during rasterization (sort-last). Sort-first means redistributing raw primitives (before their screen-space parameters are known). Sort-middle means redistributing screen-space primitives. Sort-last means redistributing pixels, samples, or pixel fragments [96]. Using any one of these choices leads to a completely different class of parallel rendering algorithms.

The pipeline in a parallel graphics system can be thought of as having two primary parts, geometry processing and rasterization (see Figure 2). Image geometry is generally parallelized by assigning each processor to a subset of the objects in the scene. Rasterization is often parallelized by assigning each processor a portion of the pixel calculations [96]. Each of these steps, depending on the algorithm, may incur redistribution costs as well. Image geometry may incur redistribution costs as volume data moves between nodes to facilitate interpolation of the assigned points, while rasterization may occur costs as local images are moved to facilitate their combination into a complete image [109].

Sort-first In sort-first rendering, the primitives are distributed as early in the rendering pipeline as possible (during geometry processing) to the processors that

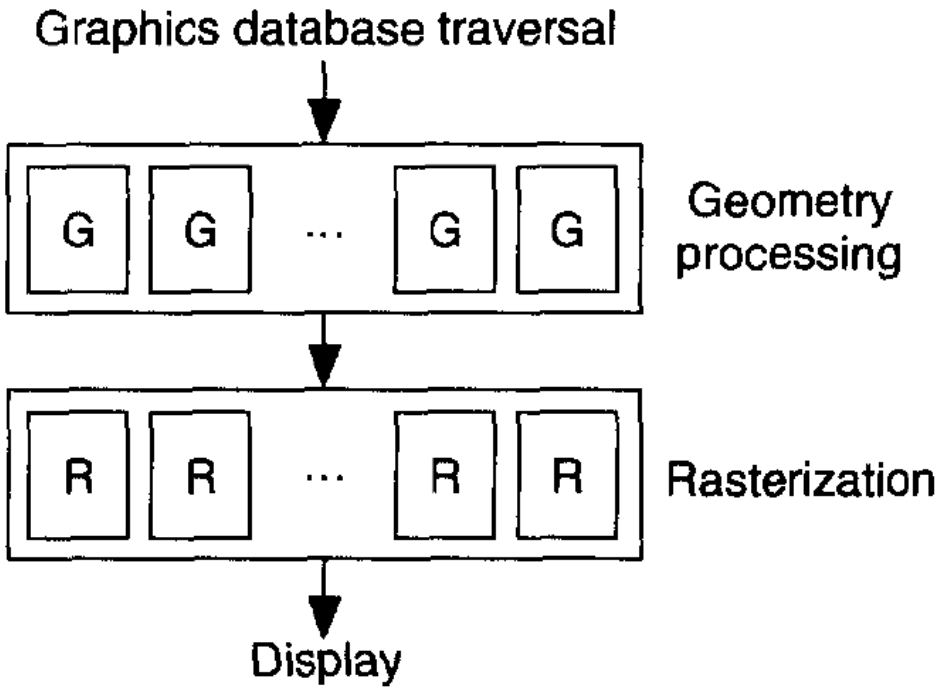


Figure 2. Graphics pipeline in a fully parallel rendering system. Processors G perform geometry processing, while processors R perform rasterization (image from [96]).

will be performing the remainder of the calculations. This method is most often used when there is a very large pixel count (as compared to geometry), as screen regions are divided among the available processors (in essence parallelizing over the screen space).

These algorithms begin with each processor being assigned a region of the screen and taking an arbitrary portion of the data, and then beginning a transformation on that data. The transformation is applied until it can be determined to which portion of the scene that primitive falls (usually calculating the bounding box [96]). Once the scene space for all of the primitives are found, those that are located on processors to which they do not belong (according to the

screen space that has been assigned to that processor), are redistributed over the network to the appropriate processors.

In summary, sort-first rendering is advantageous due to its low communication requirements when data primitives are sparse, and due to a single processor carrying out the entire pipeline for a portion of the screen. This method's drawbacks include its susceptibility to load imbalance when primitives clump into regions on the screen, giving certain processors much more work.

Sort-middle In Sort-middle rendering, the data is redistributed in the middle of the rendering pipeline. At this stage, all primitives have been transformed into screen coordinates and are ready for rasterization [96]. Each frame is first transformed by the geometry processor, and then transmitted to the appropriate rasterizer (may or may not be the same processor depending on the implementation).

The general advantage of the sort-middle technique is its straightforward implementation, and the redistribution occurs at a natural place. The disadvantages are that it can have high communication costs and is susceptible to load imbalance when primitives are not evenly distributed across the screen.

Sort-last The sort-last technique defers sorting until the end of the rendering pipeline. Each processor in this paradigm are assigned arbitrary subsets of the primitives [96]. Each of the processors computes pixel values for its subsets, irregardless of where they fall on the screen. This means that this algorithm scales well and gets a performance boost through the utilization of more and more processors [142]. At the end of the pipeline, pixels are transmitted over the network to be composited and their visibility resolved. It is at this point, however, that a

bottleneck can develop. Interactive or real-time applications which rely heavily on the network to transmit all of the pixel data will suffer in performance due to the distributed pixels. Depending on the algorithm's implementation, this can be a major drawback in sort-last techniques.

In general, sort-last parallel rendering is the only proven way of parallel rendering at scale. This is mainly because the full rendering pipeline is carried out by individual processors until pixel merging is required. In addition, this approach is less prone to load imbalance. One disadvantage, however, is that the performance of sort-last parallel rendering drops sharply as the resolution of the display increases [104]. Furthermore, the final compositing step is generally regarded as *the* bottleneck for sort-last algorithms, so methods reducing the prevalence of this bottleneck will be of great value to scientific visualization at scale [101].

Optimized Algorithms for Sort-Last Rendering With sort-last rendering being the widely accepted choice for performing image compositing at scale, a lot of work has been done in creating algorithms in this space that are highly efficient. In this section, we will list a few of the most well known and used algorithms, as well as look at a piece of open source software that integrates some of the most recent advances in compositing algorithms.

Direct Send In sort-last parallel rendering, the hardest task is the final image composting. Generally, n rendering channels will generate n full-size partial images, containing color and potentially depth [47]. These images must then be merged to form the final rendering. Direct send compositing divides the final image gathering task into n screen-space tiles to avoid exchanging full size images between the n processes. Each of the tiles is associated to a single channel for compositing, and at

the end of the compositing process all of the partial tiles are assembled to form the final image.

Another strength of this algorithm are the number of synchronization points required. In this algorithm, only two synchronization points are needed, meaning less communication overhead on the system. Communication in this method does become a problem with larger geometries. The amount of data that must be transferred across the network is proportional to the rendering resolution as the pixels from each of the sub images must be sent across the network and finally composited. This process is particularly slow when using the TCP/IP stack. Eilemann suggests that this bottleneck can be reduced by using faster network technologies such as tunneling or asynchronous transfers [47], but the overall data transfer in this scenario still remains high, and as resolutions and data set sizes increase at a much higher rate than network speed, this bottleneck becomes a major obstacle.

Binary-Swap The binary-swap method is an efficient and simple compositing algorithm that repeatedly splits the sub-images and distributes them to the appropriate processor for compositing [127]. At every compositing stage, all processors participate by being paired with another processor, splitting their image plane in half, and each one taking responsibility for one half of the plane. This means that this method will take exactly $log(n)$ compositing stages to complete.

The idea behind binary-swap, is that only non-blank pixels affect the composited results, meaning that binary-swap exploits the sparsity of the sub-images by creating a bounding rectangle that exactly encompasses the non blank region in an image. The determination of this bounding rectangle takes $O(A)$ time, where A is the number of pixels. Most importantly however, once all of the

bounding rectangles are determined, it only takes $O(1)$ time to merge two bounding rectangles, making updates to the bounding box of the composited image very efficient [127].

The primary problem of this method is the occurrence of load imbalance. Load imbalance may occur when the split of an image takes place in such a way that paired processors are given grossly different amounts of work. This means that one of the processors will have a much larger run time compared to its mate.

2-3 Swap At its core, the 2-3 swap image compositing algorithm is a generalization of a binary-swap to an arbitrary number of processors [146]. This algorithm is derived from the observation that any integer greater than one can be decomposed into a summation of a list of twos and threes, meaning that the initial partition of processors in this algorithm can be done using combinations of twos and threes. In fact, it follows that if the number of processors is a power of two, then 2-3 swap essentially becomes a binary-swap in execution stage.

This algorithm is initially started by creation a tree of the number of given processors. Each non-leaf node in this tree has either two or three children, which determines the groups of processors during each stage of the image compositing algorithm. The initial work is evenly distributed among M participating processors in a group.

The primary pros of the 2-3 swap algorithm are that it is highly flexible and can utilize any number of processors for compositing, and each processor participates in all stages of compositing, giving maximum resource utilization.

Radix-K Radix-K is a configurable algorithm for parallel image compositing [114, 65]. A unique aspect of Radix-K is its ability to overlap communication and

computation, making this algorithm very customizable to the underlying hardware of a system.

In general, the Radix-k algorithm for image compositing builds on the previous contributions of binary-swap and direct send. By parameterizing the number of message partners in a round, it unifies these two algorithms by factoring the number of processes into a number of rounds with a separate radix for each round [114].

Improving Compositing Performance with IceT Of the previous four algorithms, Radix-K is the leader in terms of work division. This algorithm performs highly parallel computation in conjunction with communication. The worst algorithm in terms of work division is direct send. Direct send is highly susceptible to work imbalance and suffers when it comes to having to communicate much larger segments to the final image. 2-3 swap and binary-swap are also susceptible to work imbalance, with 2-3 swap being more resilient. However, as stated previously, as Radix-K is able to communicate while running computation asynchronously, it mitigates imbalance and uses it to its advantage.

IceT, a leading production-quality image compositing framework, takes the problem of image compositing a step further, creating a testbed for enhancing these and other leading edge image compositing algorithms [101]. In this work, Moreland et al. found that not only were they able to create a testing ground for many different compositing algorithms simultaneously, but further, they were able to drastically improve compositing algorithms (Radix-K especially) while efficiently scaling to 64K cores. Their work demonstrates that image compositing still has room for improvement, and that through works like theirs, image compositing may

soon scale efficiently for exascale sized runs. For more discussion of the challenges of scaling visualization tasks to exascale, see Section 4.2.8.

2.3 In Situ Visualization

The total amount of data that a supercomputer can generate with a simulation far surpasses its ability to write all of that data to persistent storage. For example, Figure 3 shows the current relative bandwidth of the total compute capability of the Titan supercomputer at Oak Ridge National Laboratory versus its storage bandwidth. The five orders of magnitude difference between the two demonstrate the intractability of writing all scientific data to disk prior to performing visualization. This reality demonstrates the need for in situ on current and future machines, as the problem is only worsening.

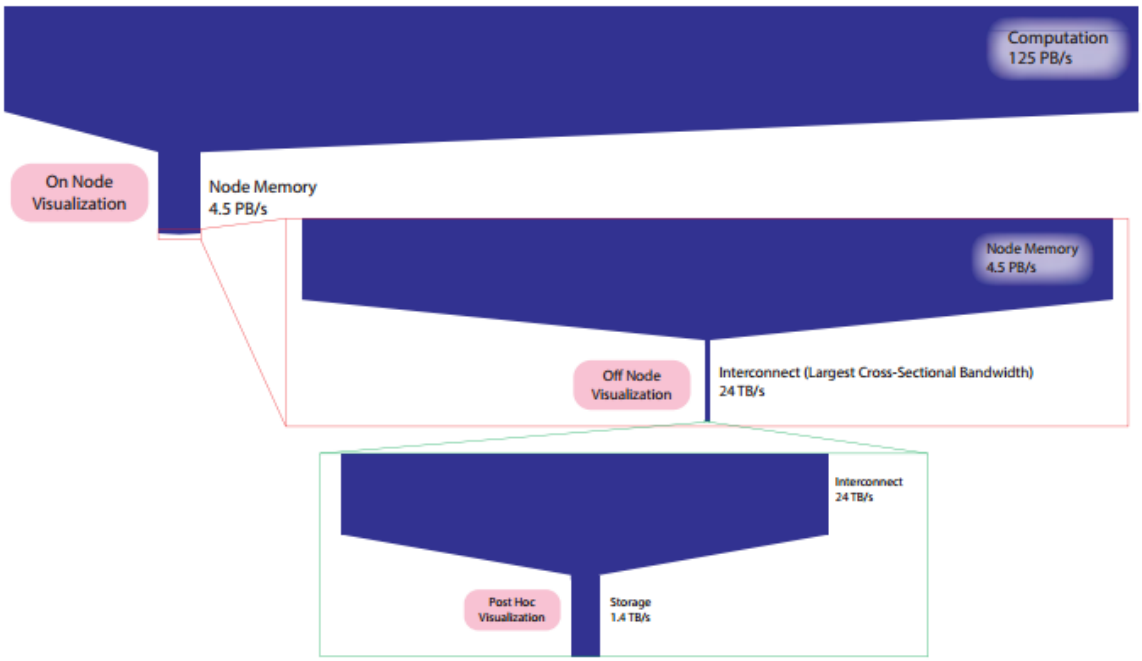


Figure 3. A plot of the relative bandwidth of system components in the Titan supercomputer at the Oak Ridge Leadership Class Facility. The widths of the blue boxes are proportional to the bandwidth of the associated component. Multiple scales are shown to demonstrate the 5 orders of magnitude difference between the computational bandwidth and the storage bandwidth (adapted from [20]).

In situ as a technology is not new, with the earliest production-quality in situ graphics being seen as early as the 1960's [18]. Therefore, it is not surprising that several past surveys of in situ and in situ techniques have been published. In 1998 Heiland et al. [62] presented a survey of co-processing systems, which covered some of the basic use and availability of predominant co-processing frameworks. A year later in 1999, Mulder et al. [107] surveyed predominant computational steering environments, whose roots lie in in situ visualization and analysis. Recently in 2016, Ayachit et al. [18] and Bauer et al. [20] present two different takes on the state of in situ technology and challenges, as well as discussions of in situ frameworks. This section builds on the ideas presented in those surveys, and presents current in situ terminology, challenges, frameworks, and in situ research covering different motivations and use cases for in situ.

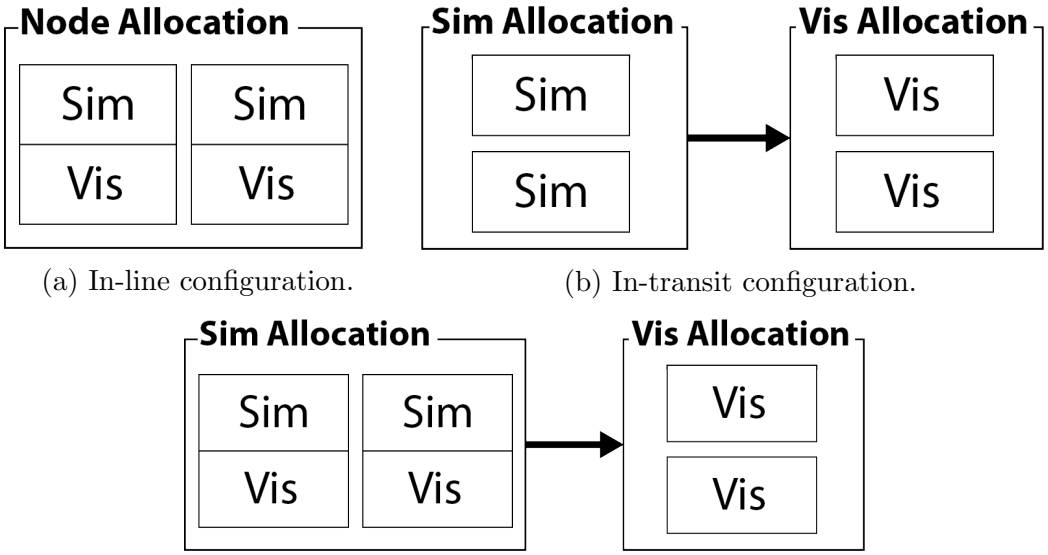
2.3.1 In Situ Terminology. In situ visualization is an umbrella term used to describe many different visualization configurations where the visualization and analysis routines are run while the simulation is still in progress, reducing the amount of data that must be transferred over the network and saved to disk [87]. The visualization community has played fast and loose with the term in situ, and it has come to mean many different things. Current efforts are underway to bring the visualization community all onto the same page about terminology, with an effort termed the "In Situ Terminology Project." The terminology being developed in this report will go a long ways towards clarifying the meaning of in situ terms for the community and our stakeholders, but will not be presented here as the report is still under development. Instead, I will stick with the more loose and general terms currently in use by the community, and will make the switch to the new terminology set as it is introduced to the larger visualization community.

The terms I will stick to in this section are as follows:

- **In situ:** Umbrella term used to describe all different types of in situ setups.
- **In-line:** In this dissertation, we define in-line to mean when the simulation and visualization code run in the same process using the same resources as the simulation.
- **In-transit:** In this dissertation, we define in-transit to mean when the simulation transfers data over the network to a separate set of visualization nodes for processing.
- **Hybrid Coupling:** In this dissertation, we define hybrid coupling to mean when there are visualization components being run on the same process as the simulation and data is still being transferred over the network to separate visualization processes on a separate set of visualization resources.

For simplification as shown in Figure 4, we view the in-line and in-transit paradigms as on-node and off-node respectively. In-line coupled can be thought of as running on the same node as the simulation, and not utilizing asynchronous data transfers from the simulation to the visualization routines, while in-transit can be viewed as on-node. Now that the definitions of in situ have been presented, we will present an overview of the challenges of using in situ techniques, and its barriers to adoption by the simulation community.

2.3.2 In Situ Challenges and Opportunities. It has only been recently that some scientists have begun to see the need to adopt the in situ approach for visualization and analysis of large-scale simulations [87]. This hesitancy is due to essentially three primary factors. First, the traditional paradigm of post-hoc visualization has meant that scientists rarely had to use supercomputer



(a) In-line configuration.

(b) In-transit configuration.

(c) Hybrid coupling configuration.

Figure 4. Simulation and visualization resource configurations for three different types of in situ.

time to perform their visualizations. The in situ paradigm would break this tradition, and scientists see visualization as a new cost and overhead to their science. Second, integrating in situ into a simulation has the potential to be a monumental task. In addition to the integration costs, the overhead of having visualization routines packaged into the simulation code in the in-line case can cause dependency issues between the simulation and visualization routines, while also bloating the size of the simulation binary. An additional side effect of this integration is the sharing of memory between the simulation and visualization routines, which can cause contention on compute nodes. Third, and the most challenging problem with in situ, is the need to know what to visualize a priori. That is, with in situ, it is required to know what to visualize including regions, values, as well as the type of visualization before the simulation starts.

These problems may seem daunting at first glance, but the issues associated with each can be mitigated through different in situ implementations. Not all in situ work has to be about visualization. In fact, a great strength of in situ methods is the ability to access all of a simulation's data during the course of a simulation, and only save what is interesting. This means in situ is a great tool for visualization, but also for data manipulations such as data reductions, explorable feature extractions, simulation monitoring, and the generation of statistics [88]. Some example work in this area includes reducing data output to an alternate explorable form, computing collections of images, and storing images enhanced with fields and meta data for post hoc exploration.

An example of creating a reduced alternate data form is by Agranovsky et al. [11]. They describe a novel process for improved post hoc data exploration using particle advection. Instead of saving out vector fields every *nth* iteration, a basis trajectory is saved. A basis trajectory is a snapshot of a particle movement between the saved snapshots. This means that a representative set of particles are traced in situ while the simulation runs, and their trajectories are output. This technique allows for new particle trajectories to be interpolated between known trajectories, increasing both speed and accuracy.

Examples of computing collections of images for post hoc exploration comes from Yen et al. [143], Chen et al. [34], and Ahrens et al. [15, 16]. Yen et al. enable post hoc interaction with images through lighting and color transfer function changes, performing slices, and changing view. Chen et al. take the approach of visualizing a large sampling of possible visualization configurations in situ (various isocontour levels, different views, etc.), and then providing an interface to explore the collection interactively. Ahrens et al. take the approach of saving many images

from many angles from a simulation instead of writing simulation data to disk. The system is called ParaView Cinema. The idea is that if hundreds or thousands of images are created for a given time step, that it will be possible to create an interactive database for a time step that will allow interactive exploration much like that of VisIt or ParaView. In addition, this system has the capability of recreating a facsimile of the surface of the data based on the many saved images, letting different color maps and scalar fields be applied to the images during the post hoc exploration. The ParaView Cinema approach was demonstrated using a large-scale model for prediction across scales ocean simulation, and it was shown that the interactive database could be generated at twice the cost of generating an equal number of traditional in-line in situ images. This cost may seem high, but the interactive database has a lot more functionality than a traditional image, allowing for the greater flexibility of post hoc exploration.

Finally, examples of generating images with enhanced meta data for post hoc exploration comes from Tikhonova et al. [132, 133] and Fernandes et al. [51]. Tikhonova et al. describe a method of storing layers of isosurface images that could later be composited together for post hoc exploration. Fernandes et. al. used a similar technique for volumetric renderings (saving areas of interest along with depth information) that could be explored post hoc.

The following three sections will present more in depth information about the three in situ techniques. They will discuss the strengths and weaknesses of each technique, provide a look at in situ frameworks in those categories, and give examples of past works performed using each paradigm to motivate the technique.

2.3.3 In-line In Situ. With in-line in situ, in situ routines will directly share the same resources as a simulation. This has many different

advantages and disadvantages. By sharing the same compute nodes, the simulation and visualization codes compete for memory, making the careful design of in situ routines critical with in-line in situ. An inefficient or buggy implementation could slow the simulation, or worse, cause it to crash.

Further, by sharing the same resources, the in situ routine will be required to operate on the same level of concurrency as the simulation, which could cause slow performance with some in situ routines. Moreover, with this approach, the simulation code must wait for the in situ processing to complete after each simulation time step before it can carry on with computation [118]. This lock-step approach to computation is not attractive to many simulation scientists, which is part of their angst against using in situ techniques.

Even with these potential issues with in-line in situ it is a widely used technique, with many different works taking advantage of data locality and computing power available to a full scale simulation.

2.3.3.1 In-line In Situ Frameworks. This section presents a look at in-line in situ frameworks, and explores their features and restrictions. While many in situ frameworks have the potential to operate in several different modes, the frameworks presented here either operate fully or primarily in the in-line model. For each framework we give a short description of functionality and categorize them according to the in situ methodologies they employ.

Cactus Cactus [56, 1] is a development environment in which an application can be developed and run. In addition, Cactus has the capability of instrumenting legacy codes, to prevent the need for redesign within the Cactus framework. The remote visualization and data analysis capabilities of Cactus are achieved with in-line in situ. Visualization operations are performed on the computational

nodes, and the resultant geometry can then be sent to a remote viewer or saved to disk. An additional capability of Cactus, is that computational steering can be accomplished through the remote viewer, on predefined variables in the instrumented code.

CUMULVS The Collaborative User Migration User Library for Visualization and Steering (CUMULVS) [67] is an infrastructure to allow multiple users the ability to monitor and steer of a simulation remotely. Users can connect and disconnect at will during the course of the running simulation. CUMULVS is capable of text and 2D output and visualization from an instrumented simulation. The original 2D visualization was supported through the use of AVS. A downside of the CUMULVS system is that it does not support the output of images, graphics are used purely for simulation monitoring.

ParaView Catalyst ParaView Catalyst [50, 19] is the ParaView library which allows for in situ visualization of simulation output using the full visualization feature-set of ParaView, or subsets of features, by using reduced size binaries when minimal memory overhead to the simulation is required. Catalyst operates in a in-line fashion, pausing the simulation while data operations take place.

Catalyst also allows for simulation steering and monitoring by connecting the Catalyst routines instrumented into the simulation to the ParaView application. This is a powerful feature that allows researchers to step through their code and dynamically modify visualizations based on the progress of the simulation.

In order to use Catalyst it must be instrumented into the simulation code, and an adapter needs to be written to define the interface between the simulation and Catalyst. This adapter defines how the simulation can call Catalyst as well as

maps the simulation data to the VTK data model used by Catalyst. Catalyst has proven to be highly scalable, with the current largest run being on 256 thousand cores.

Ascent Ascent [63, 77, 78, 79] is a system designed to explore in situ visualization and analysis needs for science codes on exascale architectures. An additional use for the infrastructure is as light weight prototyping environment for in situ analysis and visualization routines. This prototyping environment allows for fast implementations of in situ ideas. It uses Conduit [3] for a data model, VTK-m for the visualization and analysis pipeline, and IceT [98] for parallel image compositing.

Ascent supports execution on many core environments, multiple programming languages, and works within a batch environment. Additionally, it supports zero-copy of the data when possible. Ascent has been extended in recent years to support experimental in-transit work, as well as extensions to other infrastructures such as Cinema, Jupyter notebooks, and VisIt.

VisIO VisIO [95] is an I/O library for use on distributed file systems within visualization applications. It includes a new scheduling algorithm to help preserve data locality within a simulation by assigning visualization intelligently to co-locate computation and data. The core of this framework revolves around the use of the Hadoop distributed file system in conjunction with a VisIO enabled reader in ParaView. One drawback of this approach is that it requires the use of the Hadoop file system, which could prove very time consuming to use in an existing application.

VisIt Libsim VisIt Libsim [139] is the VisIt library which allows for in situ visualization of simulation output using the full visualization feature-set of VisIt. Libsim operates in a in-line fashion, pausing the simulation while data operations take place. In fact, when the Libsim library is inserted into a simulation program, it makes each process of the simulation act much like a VisIt compute engine, operating in the same data space as the simulation.

One interesting feature that stems from the engine-viewer approach used in VisIt, is that the Libsim routines within the simulation listen for a request to connect by a VisIt process, meaning that users can connect and disconnect from the in situ routines as needed to perform periodic simulation steering or to check validity.

One drawback of the Libsim approach is that it requires instrumentation of the simulation code. Several calls need to be inserted into the simulation, as well as the Libsim binary itself. In some cases, if a simulation does not have a well-defined loop to simulate a single time step, Libsim suggests restructuring of the simulation code.

Nevertheless, Libsim remains a powerful in situ visualization tool, largely due to the large array of visualization capabilities within the VisIt tool itself. It has also been shown to scale well, nearly as well as VisIt itself, up to 62 thousand cores [140].

2.3.3.2 Related Work: In-line In Situ.

Implementations using in-line in situ are often concerned most with the full utilization of a resource. That is, the desire is to run the simulation at the largest capacity possible, not reserving nodes for visualization or I/O. This implementation does have the advantage that the visualization routines have direct access to the full simulation output, and the

full parallel capacity of the simulation machine. The following are several works that utilize in-line in situ for visualization.

Yu et al. [145] demonstrate an in-line system for volume rendering of jet fuel combustion data, in addition to a remote viewer application used to view the volume rendered images during the simulation run, as well as send requests for different viewing angles or transfer functions to the simulation code. The visualization code in their case was directly integrated into the simulation code, and worked off of pointers to the simulation results in order to reduce data duplication. As this system required the simulation to pause while visualization was taking place, it had a large effect on simulation runtime, with combined visualization and I/O times (from compositing) taking up to 4x more time than the simulation when done at every time step. This was reduced to two orders of magnitude less than the simulation time though, when the temporal fidelity was dropped to every ten time steps.

Woodring et al. [141] describe an in situ workflow for saving a simulation-time random sampling of large-scale particle datga from a cosmological simulation. Their workflow uses an extension of the kd-tree stratified random samping algorithm to generate level-of-detail output files for post hoc visualization. The level of detail approach is used in order to reduce storage bottlenecks and give them an integrated approximation error for their views. Using the kd-tree approach they are able to tune the output size to their specific needs by changing how many levels of the tree are written to disk, and show that at the lowest level of detail that they can write only 1/64th of the total simulation data to disk. This approach is useful in that it still allows for exploration of the data post-hoc, which is advantageous to static images.

Lorendeau et al. [84] describe a workflow using the Catalyst in situ visualization library for visualizing a computational fluid dynamics code. Catalyst is a ParaView library that defines in situ workflows using parallel VTK. In the described workflow the authors developed an adapter to their simulation worfklow for Catalyst and use it to perform their visualization operations. By introducing Catalyst they were able to perform their visualization operations in situ and save on the amount of data written to disk. They saw a 20 to 30% overhead associated with their initial implementation, but predict it can be reduced with better memory management in their adapter.

2.3.4 In-transit In Situ. In-transit in situ offers many new configurations for visualization not seen with in-line in situ. The most common configuration is to have a set of dedicated visualization nodes on the same machine as the simulation, which reduces the effects of network latency that is seen when moving data to another machine. This separate allocation allows the visualization routines to run concurrently with the simulation, not impacting its runtime as with in-line methods.

This benefit of a separate set of visualization nodes is also a primary downside of in-transit visualization, as simulation scientists rarely want to give up portions of compute power for visualization tasks. Recently however, it has been shown that by streaming simulation data to an allocation of staging nodes, that the effects of disk latency can be hidden by staging the disk writes to the separate allocation, and letting them run while the simulation continues [10, 110, 102]. Given this, the approach of dedicating a set of the simulations nodes to staging becomes more palatable to simulation scientists, and further allows the introduction

of in situ visualization techniques on that separate allocation, which can further benefit the simulation.

2.3.4.1 In-transit In Situ Frameworks.

This section presents a look at in-transit in situ frameworks, and explores their features and restrictions. While many in situ frameworks have the potential to operate in several different modes, the frameworks presented here either operate fully or primarily in the in-transit in situ model. For each framework we give a short description of functionality and categorize them according to the in situ methodologies they employ.

EPIC The Extract Plug-in Components Toolkit (EPIC) [46] is designed to create in situ data surface extracts from a running simulation. These extracts can be viewed in situ using a prototype version of FieldView, or extracts can be saved to disk. One downside of EPIC is that it requires the simulation to use the EPIC defined MPI communicator. This requirement could cause substantial integration issues for codes wishing to employ EPIC.

Freeprocessing Freeprocessing [52, 7] is an in situ interposition library designed to reduced the barrier to entry for simulations to introduce in situ visualization. The premise is that many visualization codes avoid in situ technology as it has a large upfront cost for integration, and worse, if it requires direct manipulation of the simulation source code, it could have negative repercussions for performance and code stability. Freeprocessing has the ability to do in-transit visualization using staging nodes, in either a synchronous or asynchronous mode. Further, Freeprocessing can connect to existing visualization tools such

as VisIt Libsim or ParaView Catalyst to take advantage of existing work in high performance visualization routines.

ICARUS Initialize Compute Analyze Render Update Steer (ICARUS) [118] is a ParaView plug-in for in situ visualization and computational steering. It operates in the in-transit in situ environment using a shared memory mapped HDF5 file for data access. It has minimal modification requirements for a simulation code, but only operates on the HDF5 file format. Simulation steering is accomplished through the use of the shared file interface, where each side can read and write from the files to pass steering messages.

pV3 Parallel Visual3 (pV3) [61, 60] is a parallel visualization system primarily targeted at computational fluid dynamics codes. It utilizes a client-server architecture, and has built in visualization capabilities. The client-server architectures allows the system to connect to an instrumented simulation at will. The pV3 system allows for computational steering, in-line, and post-hoc visualization. pV3 is no longer under development.

2.3.4.2 Related Work: In-transit In Situ.

Past works that utilize in-transit in situ are most often concerned with the impact that visualization has on a running simulation. Works in this category often try to reduce the effect that visualization has on the simulation time as much as possible, and often do so by running on a separate allocation. The following are several different approaches to in-transit visualization.

Ellsworth et al. [48] describe a time-critical pipeline for weather forecasting using the GEOS4 simulation code. This code is run under very tight time constraints four times a day, which requires the visualization to be performed

with minimal overhead. The visualization is achieved in this workflow by copying the simulation data to a separate shared memory segment where a discrete visualization system then accesses and operates on the data. This setup does require that the simulation be instrumented, and several new calls had to be added directly to the simulation code to redirect the output to the desired shared-memory segment. The resultant time-varying visualizations are then saved to disk or displayed on a tiled wall display.

Ma et al. [88] describe a visualization system for an earthquake simulation that uses a remote viewer over the wide are network to interactively change the visualization operations, view angles, color, etc. of rendering operations being done on the simulation machine itself. The integration of their visualization system requires that a simulation provide an API to access the internal data structures of the simulation, so the integration is visible from the perspective of the simulation scientist. However, this approach does limit the amount of integration needed compared to other more intrusive methods. The authors then demonstrated the viability of their system by interactively visualizing the results of a 2048 process simulation.

Pugmire et al. [116] introduce a visualization workflow that utilizes ADIOS to intercept the I/O calls of a simulation and stage the simulation data on a separate allocation of nodes. Their workflow then used EAVL to perform parallel visualization operations on the staged data, Mesa [8] to perform rendering, and IceT to perform parallel image compositing. Their experiments show that by incorporating Mesa and IceT into the parallel visualization environment EAVL, that they were able to further reduce the time to completion by between 5% and 14% versus an MPI compositor.

2.3.5 Hybrid In Situ and Computational Steering.

Hybrid methods [38] are composed of both in-line and in-transit components being utilized simultaneously. These methods support the flexibility of processing and reducing data on the simulation resources before they are either written to disk, or transferred to the visualization resource for additional processing. In other words, it offers the ability to achieve the best of both the in-line and in-transit paradigms.

Computational steering systems are methods related to hybrid in situ, as they allow a user to control all aspects of the computational science pipeline [64]. This control can range from simple monitoring controls to check that a simulation is in a valid state, to advanced controls that allow a user to step through a simulation and change key simulation variables while a simulation is in progress. One advantage of computational steering is that it can enable a user to steer a simulation back to a valid state, or stop an invalid simulation before computing time is wasted on invalid computations.

2.3.5.1 Hybrid In Situ and Computational Steering Frameworks.

This section presents a look at hybrid in situ frameworks, and explores their features and restrictions. For each framework we give a short description of functionality and categorize them according to the in situ methodologies they employ.

ADIOS The Adaptable I/O System (ADIOS) [82, 66], is a componentization of the I/O layer used by high-end simulations and/or for high-end scientific data management, providing an easy-to-use programming interface, which can be as simple as file I/O statements. ADIOS abstracts the API away from implementation, allowing users to compose their applications without detailed knowledge of the underlying software and hardware stack. The ADIOS framework

has been designed with a dual purpose: to increase the I/O throughput of simulations using well-known optimization techniques, and also to serve as the platform for introducing novel data management solutions for production-use without extensive modifications to the target applications.

ADIOS is used by a variety of mission critical applications running at DOE and NSF facilities, including combustion, materials science, fusion, seismology, and others. At the same time, ADIOS offers the community a framework for developing next generation I/O and data analytics techniques. Recent advances in this area include FlexIO [149], an infrastructure for the flexible placement of in situ analytics at different levels of the memory hierarchy, and PreDatA [148], a strategy for characterizing data while it is being generated in order to support faster data manipulations on staging resources.

To address the growing imbalance between computational capability and I/O performance, ADIOS introduced the concept of data staging, where rather than writing data directly to shared backend storage devices, a staging pipeline moves data to a transient location, on separate physical nodes and/or on memory resources on the same node where data is generated. Once on the *staging* nodes, data can be aggregated, processed, indexed, filtered, and eventually written out to persistent storage [30]. A key outcome of staging has been dramatic reductions in the total volume of data to be stored through the use of in-line and in-transit data analytics. ADIOS contains a variety of transport methods for the movement of data, including DataSpaces [43], which allows memory coupling between processes running on different sets of nodes, FlexPath [42], which supports a publish/subscribe interface for direct memory access, and ICEE [41] which supports RDMA transfers over wide area networks.

Damaris/Viz Damaris/Viz [45, 5] is an in situ framework based off of the I/O middleware framework Damaris [4]. Damaris/Viz was developed with the goals of having low impact on simulation runtime, low impact for in situ integration, and high adaptability. It achieves these goals by having low instrumentation costs. Visualization capabilities consist of user-defined modules, or connections to the VisIt Libsim or Paraview Catalyst interfaces. Damaris/Viz can operate in either an in-line approach, utilizing a subset of cores on each simulation node, or in-transit, by using a dedicated set of visualization nodes.

EPSN EPSN [6] is a library designed to provide a software environment for computational steering. There are two methods of interacting with EPSN, a lightweight network user interface, or through a distributed parallel visualization tool. The visualization and steering tools utilize VTK and IceT. EPSN has a client server relationship allowing multiple clients to connect and disconnect to the simulation on-the-fly.

GLEAN GLEAN [136] is a non-intrusive framework for real time data analysis and I/O acceleration. It achieves this by being semantically aware of the data it is transporting, and by mitigating the variability of filesystem I/O performance through asynchronous data staging nodes using the network. GLEAN follows a similar model to ADIOS, and allows for custom data analyses to be performed on both the compute and staging resources. This model can mitigate the overall data saved to disk, improving application performance.

GLEAN supports both the in-line and in-transit in situ paradigms. In-line workflows are supported when GLEAN is embedded as part of the simulation, sharing the same address spaces and resources, and the simulation is semantically

aware when it calls GLEAN. In-transit workflows are supported when GLEAN asynchronously moves simulation data to a separate allocation of staging nodes though standard I/O libraries like HDF5.

Numerous performance studies exist using GLEAN, and it has been shown to be scalable and has drastically improved I/O performance on test codes that traditionally used HDF5 or pnetcdf. Overall, GLEAN is a powerful framework that requires minimal or no modifications to existing applications to implement, and can improve application performance on applications experiencing network bottlenecks. That is, simulation scientists can focus on simulation development, and let GLEAN focus on data transport enhancements, while also giving the simulation new opportunities to insert data analysis methods on both the simulation and data staging nodes.

Magellan Magellan [135] is a framework for computational steering of a simulation. To instrument a code with Magellan it must be annotated to reveal specific steering parameters to the Magellan interface. This interface consists of two components, steering servers and steering clients. The steering client is a mechanism to interface with the steering servers and interactively change parameters. Magellan allows for multiple applications to be steered simultaneously, but is very limited in its graphical capabilities. It must be linked with outside visualization systems for the creation of visualizations. Magellan is no longer under development.

SCIRun SCIRun [112] is a programming environment that allows for the construction, debugging, and steering of scientific computations. The computational steering aspect of SCIRun is one of its more highly developed

aspects, allowing users to vary different aspects of a simulation while it is running. This interactivity is performed lock-step, so it follows the in-line approach of stalling the simulation while it performs its steering and analysis. SCIRun is modular, so further extensions can be added through the modular interface.

SENSEI SENSEI [18, 57, 85] is an effort to both streamline the in situ instrumentation of a scientific code and allow for flexibility in the choice of analysis infrastructure. This flexibility is achieved through the use of the underlying technologies that SENSEI employs. It allows for the use of VisIt Libsim, ParaView Catalyst, and Ascent as visualization platforms, and GLEAN, HDF5, or ADIOS for data staging. The analysis routines in SENSEI use the standard VTK data model for cross-platform compatibility.

SENSEI has even addressed some of the drawbacks of the VTK data model discussed earlier in section 2.2.1, by adapting the VTK data model to support structures-of-arrays, array-of-structures, and zero-copy.

To instrument a code with SENSEI, there are two adapters that need to be created. First, a data adapter API is created. This adapter is used to provide the analysis code with access to simulation mesh and array attributes. Second, an analysis adapter API is created. This adapter provides a concrete instance of an analysis adapter, which is a mechanism for interfacing with different in situ infrastructures. Figure 5 gives an overview of possible SENSEI instrumentation layouts. It is possible to perform both in-transit and in-line analysis with this interface, with multiple options for staging and visualization technologies.

2.3.5.2 Related Work: Hybrid In Situ. Past work in the area of hybrid in situ and computational steering often focus on making in situ more accessible to simulation teams, providing greater temporal locality of simulation

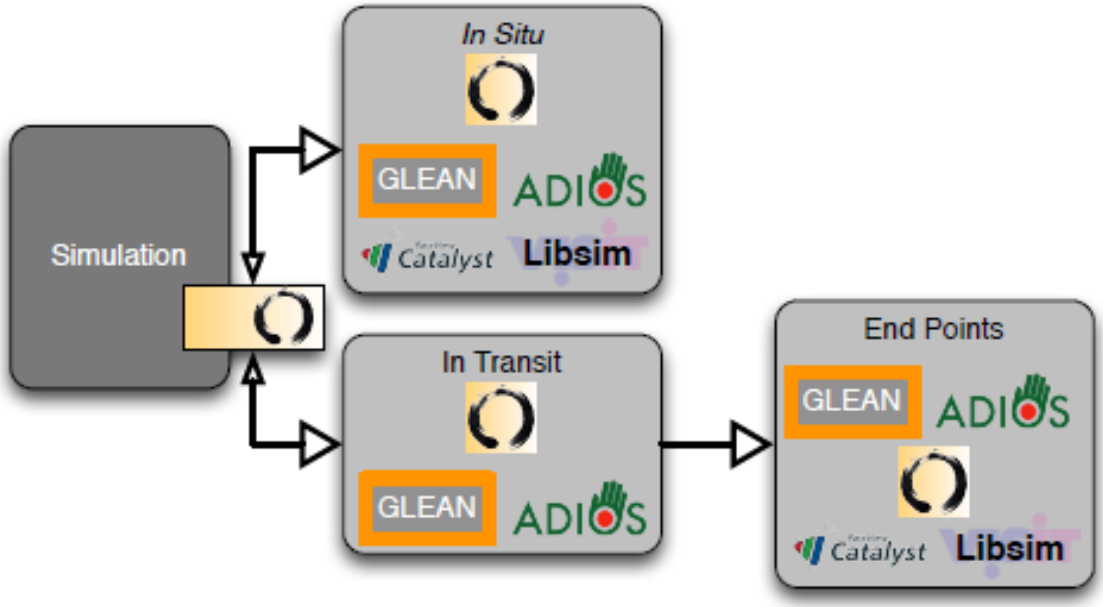


Figure 5. A depiction of the SENSEI generic data interface for in-line, in-transit, and hybrid implementations. It enables the dynamic choice of instrumentation technology depending on user circumstances though the use of its generic interface (adapted from [18]).

visualizations, and providing a channel for the simulation team to interact with the running simulation directly. Some of the works presented below take advantage of the different frameworks presented above, while others roll their own approaches to specific simulation needs.

Past work in the area of simulation monitoring and steering has focused a lot of effort into designing methods for quickly and efficiently visualizing data across a network. Some notable examples include Visapult [24], Visualization Dot Com [25], VisPortal [26], and a Real-Time Monitoring framework for large scientific simulations [113]. VisPortal and Visualization Dot Com build on the foundations of Visapult, and provide a remote distributed visualization framework for efficient visualization of remote simulation data. This framework uses both the local visualization client and the remote data client to perform parallel renderings,

decreasing the time to produce the final visualizations. By leveraging Visapult, VisPortal and Visualization Dot Com are able to provide convenient access to simulation data to scientists through an easy to use and accessable online interface.

A different approach to simulation monitoring is the online dashboard. One successful instance of an online dashboard is eSimon [129], used for the XGC1 simulation. This dashboard was launched with each simulation run and was responsible for several different common visualization and analysis tasks in XGC1. First, the dashboard was responsible for creating and updating plots of approximately 150 different variables every 30 seconds and plotting 65 different planes for the live simulation. At the conclusion of a run, the dashboard would automatically output movies of each of these plots of interest for quick review. In addition, this dashboard cataloged simulation output allowing users to search for and retrieve data of interest, without having to locate and search through simulation output files. Finally, this dashboard was available to scientists anywhere in the world through their internet browsers. This approach to simulation monitoring is powerful, as it is easy-to-use from the point-of-view of the simulation scientist and is easy to access.

Moving on now to works on visualization, we look at a few works utilizing ADIOS. ADIOS is an enabling technology, and a number of past visualization works have taken advantage of the easy integration and data transfer and translation capabilities of the platform. Some recent examples include work by Bennett et al. [22], Pugmire et al. [115], and Kress et al. [69].

The work by Bennett et al. makes the insight that many analysis algorithms can be formulated to perform various amounts of filtering and aggregation, resulting in intermediate data that can be orders of magnitude smaller than

simulation output. They put this insight into practice by creating a two stage pipeline using a combustion simulation, in which data is first filtered and reduced on the simulation nodes before being transffered to a staging area using ADIOS. Once in the staging area they performed topological analysis, gathered descriptive statistics, and performed visualization. They validated this approach at moderate scale showing that it was possible and fast to perform these operations in a hybrid fashion.

The work by Pugmire et al. focused on the development of scalable visualization plugins that operate within the data staging of ADIOS. They show the creation of an interactive visualization system which utilizes the RDMA transfer capabilities of ADIOS for data transport, and VisIt for visualization. ADIOS would send subsets of data requested by the visualization client to a visualization cluster where VisIt scripts would operate on the data, with the final results being viewed by a remote visualization client. Figure 6 shows the result of a visualization using their system, which is the visualization of a turbulent eddy and its accompanying particles within the fusion simulation code XGC1.

Kress et al. focused primarily on data reduction using ADIOS and a separate analysis node allocation. Their premise is that at exascale, simulation data reduction will be required in order to gain a reasonable temporal view for visualizations. They present two different types of data reductions that can be done in staging by altering the underlying data representations. One interesting approach they present is representing data with reduced precision formats. That is, simulations are typically over-resolved, so for visualization it is not necessary to maintain full precision, and they demonstrate that visualizations are comparable at different digits of precision. They caution however, that data reduction must be

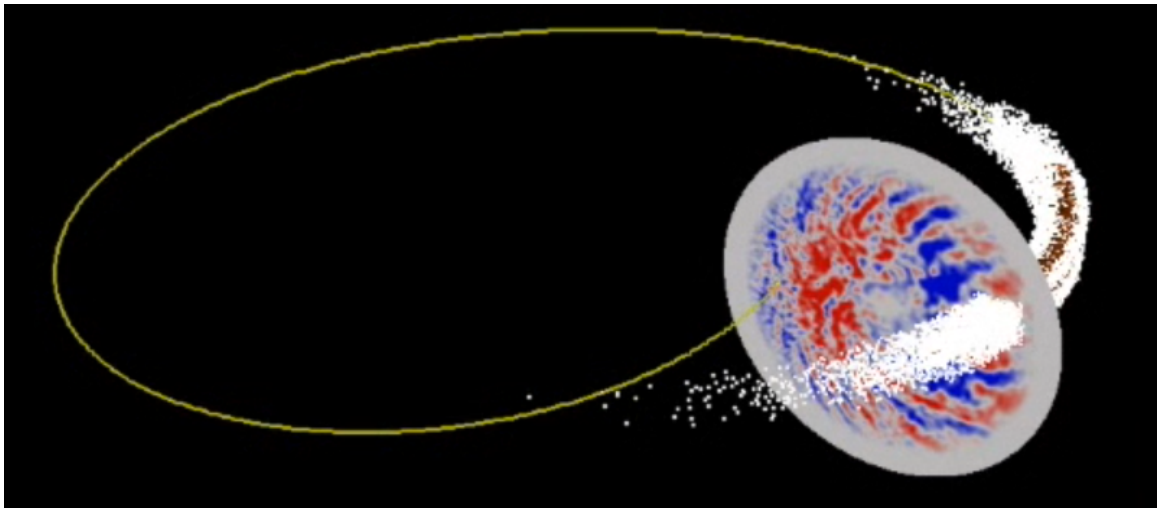


Figure 6. The VisIt interface window demonstrating particle tracking by ID of particles that were inside a 3D eddy at a particular time step in the past (from [115]).

done with domain knowledge. Data features may be lost when doing visualizations of derived variables.

A further example that does not utilize ADIOS is by Vishwanath et al. [137]. They describe a test of the GLEAN framework on an adaptive mesh hydrodynamics code, in which they increased I/O speed and computed fractal dimensions of the data as it was being written to disk. In this work, they were able to instrument the simulation code without adding anything to the simulation code itself, instead the I/O libraries already in use by the simulation were instrumented to use GLEAN. Through their tests they say that it was much faster to compute the fractal dimensions in situ versus their traditional post hoc approach, and that they were able to increase I/O speed between 10-117x vs HDF5 and pnetcdf.

A more basic example not utilizing a framework is by Buffat et al. [32]. They describe a client-server system for in situ analysis of computational fluid dynamics. Their workflow has the capability of performing computational steering,

and can use VisIt Libsim for remote visualization. The core of their workflow is a separate allocation of nodes where the visualization tasks take place in Python, and the data is asynchronously transfered to this allocation from the simulation using MPI.

2.4 Summary

This chapter provides a background and survey on the major topics that intersect with this dissertation, including high performance computing, scientific visualization, parallel graphics and its bottlenecks for scientific visualization, and in situ visualization with an emphasis on existing in situ infrastructures. This background material serves as a primer for the upcoming chapters, each of which help inform the dissertation question of *"In-line vs. in-transit insitu: which paradigm is the most efficient and under what circumstances?"*.

CHAPTER III

IN SITU VISUALIZATION NEEDS: REALITY FROM THE FRONT LINES

Most of the text in this chapter comes from [74], which was a collaboration between Scott Klasky (ORNL), David Pugmire (ORNL), Hank Childs (UO, LBL), and myself. The writing of this paper was a collaboration between Hank Childs, David Pugmire, and myself, and I performed the lead role on all writing. Hank Childs provided text edits and a sounding board for designing the survey and compiling our results. David Pugmire and I designed and conducted the user surveys. Scott Klasky was involved in initial discussions of the survey and manuscript.

In situ techniques have become a very active research area since they have been shown to be an effective way to combat the issues associated with the ever growing gap between computation and I/O bandwidth. In order to take full advantage of in situ techniques with a large-scale simulation code, it is critical to understand the breadth and depth of its analysis requirements. In this chapter, we present the results of a survey done with members of the XGC1 fusion simulation code team in order to gather their requirements for analysis and visualization. We look at these requirements from the perspective of in situ processing and present a list of XGC1 analysis tasks performed by its physicists, engineers, and visualization specialists. This analysis of the specific needs and use cases of a single code is important in understanding the nature of the needs that simulations have in terms of data movement and usage for visualization and analysis, now and in the future. We start by motivating the need to understand the specific in situ visualization needs of simulation codes, describe related work, explain the specifics

of the simulation code we surveyed, and do an in depth look at the analysis and visualization requirements collected from the survey.

3.1 Motivation

Current trends in supercomputing point to a future where increases in core counts are greatly outpacing increases in memory and I/O bandwidth. These systems will make it possible to compute far more data than can regularly be moved to disk. As a result, the vast majority of data produced by simulations will be lost, or the workflow will stall under the burden of I/O [12]. Simulation scientists are faced with the problem of deciding what small fraction of data can be saved, and what must be discarded. Ever lurking within these decisions is the possibility of lost scientific knowledge.

Research efforts for efficiently using these systems are following several paths. These paths include more efficient use of the memory hierarchy in terms of I/O [82, 130, 136] and burst-buffers [81, 120], data compression and subsetting [76, 80, 115, 150], frameworks that efficiently use the available compute cores to process data [93, 97, 103], and in situ visualization and analysis methods [46, 50, 78, 112].

In this chapter, we limit our consideration of this topic to the overall dissertation theme: in situ visualization methods. We focus our efforts on a study of the XGC1 [33] scientific team, and the workflows being run on leading edge supercomputing systems. We present a survey of the predominant visualization and analysis tasks in this workflow, and, for each, describe how the task is currently performed given a list of computational, time, and resource constraints. We believe this study of the XGC1 project is valuable, since it formalizes the specifics of in situ requirements for a simulation code for later usage by visualization scientists. While a subset of this information is available in several research papers, we think

a study dedicated exclusively to cataloging requirements gives a more complete picture. This information could in turn be used for engineering software designs, hardware designs, and conducting feasibility studies.

We know of no efforts to provide a formalized way to approach in situ visualization given the computational and data constraints and requirements of a particular simulation. Such a formalization would provide a framework to reason about the time required for input and output on a particular computing system, along with the scientific requirements for visualization in a workflow, which in turn informs the feasibility of that in situ task. While we do not solve the feasibility problem in this work, we believe that data gathered in this work will be input to solutions for the feasibility question.

In the remainder of this chapter, we discuss related works in Section 3.2, describe the XGC1 project and its output data and data sizes in Section 3.3, and describe visualization and analysis requirements for XGC1 from our interview process in Section 3.4.

3.2 Related Work

We know of no work focusing specifically on cataloging and categorizing the different visualization and analysis tasks of a simulation code. There are however instances of visualization and analysis requirements being reported in conjunction with a study.

A work by Bennett et al. [22] reports on a use case with combustion simulations using S3D, where features are tracked, identified, and visualized both in situ and in transit. Their work utilized in situ and in transit methods using a volume of nearly 1 billion cells and 16 seconds average wall time per time step using 4896 cores.

Pugmire et al. [115] explore a feature tracking and identification use case in the XGC1 simulation code, using a data set of nearly 1 billion particles and a time budget of 10 seconds per simulation time step. In this work, the authors describe a system that intelligently handles the tracking of particles and features of a simulation in real time, in a user specified area of interest.

Ellsworth et al. [48] describe a time-critical pipeline for weather forecasting using the GEOS4 simulation code. This code is run under very tight time constraints four times a day which requires the visualization to be performed with minimal overhead. The visualization was performed on data consisting of 23 million cells with up to seven 3D and four 2D fields per cell.

Malakar et al. [90] describe a series of visualization tasks done with the LAMMPS simulation code. The data contained 1 billion atoms, using 91 GB per simulation time step. Typical runs consisted of 1000 time steps, with output every 100 time steps.

Slawinska et al. [125] demonstrate the incorporation of ADIOS into Maya for an astrophysics simulation workflow. Using in situ techniques, they reduced the amount of data needed to perform their visualization and analysis task from 4.5 TB down to 24 GB that would normally be saved to disk without in situ.

From these past works we have been able to get a sense of some of the data sizes and visualization and analysis requirements from other large-scale simulation codes. None of these reports however gives a full picture of the data and analysis requirements stemming from these simulation codes. Without understanding both the breadth and depth of the needs of these codes in terms of data movement and usage, future research efforts on in situ techniques may miss an important aspect

or problem that is very important to large-scale simulation codes, but just has not been formally presented to the community.

3.3 XGC1 Project

XGC1 is a 5D gyrokinetic ion-electron particle in cell (PIC) code used to study fusion of magnetically confined burning plasmas. XGC1 is used in particular to study the turbulent region on the outer region of the plasma called the *edge*. The simulation proceeds by computing the interactions of a very large number of particles, and then depositing the particles onto a finite element mesh. The mesh, as shown in Figure 7, consists of a number of 2D planes positioned uniformly around the toroidal shape of the tokamak. The number of planes used, typically between 16 and 64, is specified by the scientists to capture the expected waveform distributions. The particles, which interact within the toroidal space of the mesh, are statistically deposited onto the mesh. This deposition step provides a statistical view of simulation, as well as helps optimize the simulation runtime.

XGC1 scientists typically run two different sizes of simulations, which we categorize as *medium* and *large*. These run sizes are defined by three factors (1) the number of compute processes; (2) the number of particles per process; and (3) the number of nodes in the mesh. These factors are quantified for the medium and large runs in Table 4.

Table 4. Simulation size characteristics, particle counts, and wall time per simulation time step for two different XGC1 run sizes.

	Medium Run	**Large Run**
Number of Processes	65,536	262,144
Number of Particles Per Process	100,000	500,000
Number of Mesh Nodes	100,000	1,069,247
Average Wall Time Per Time Step	2-4 min	5-10 min

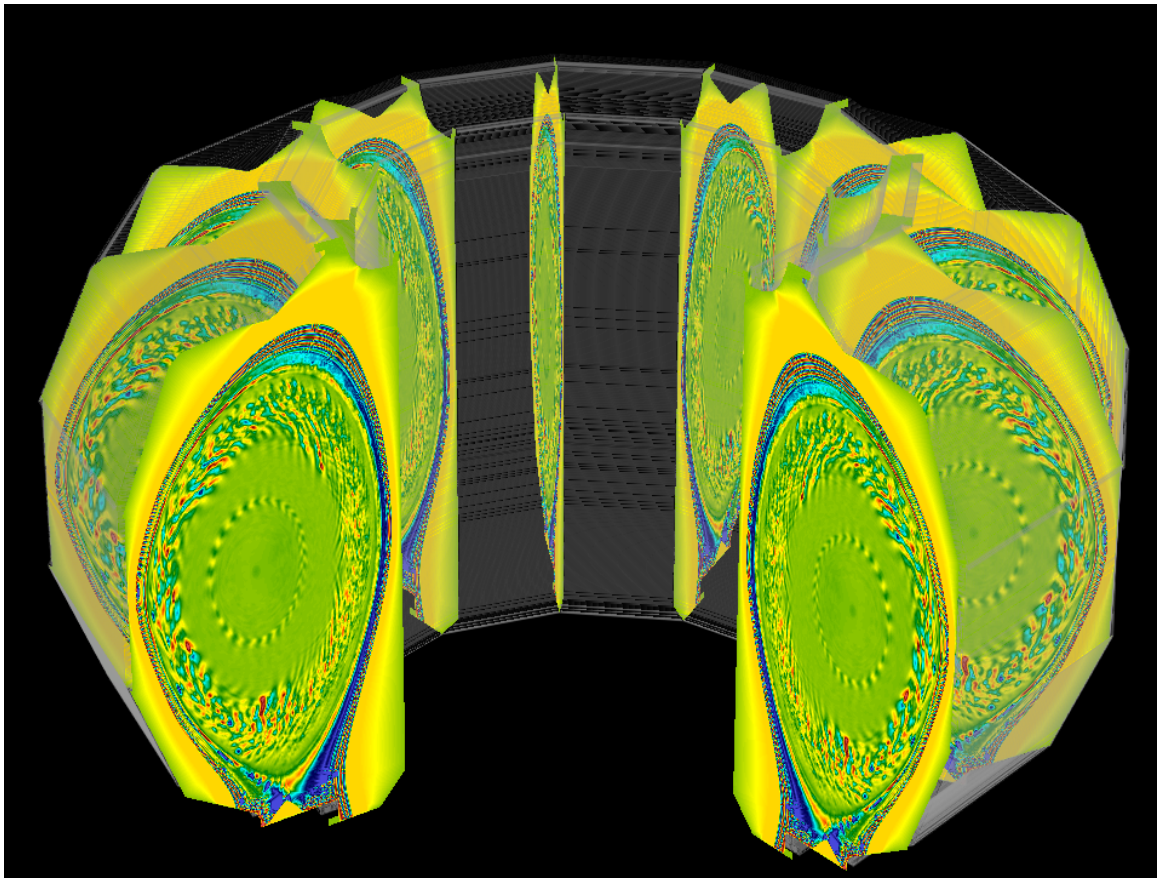


Figure 7. Example of an XGC1 mesh with planes equally spaced around the central axis of the tokamak.

3.3.1 XGC1 Output Data Types and Sizes. In this section we discuss the variety of outputs produced by XGC1, with an emphasis on outputs most relevant for analysis and visualization.

The largest output file in XGC1 is the `restart` file, and contains the state of each particle at a particular time step. Medium and large runs will contain around 6 billion and 150 billion particles, respectively.

The second largest output file in XGC1 is the `restartf0` file, which is used for post-processing detection of abnormal particles. This file contains a mapping of each plane in the unstructured grid to a regular mapping in phase space. This

mapping produces smooth contours for non-turbulent particles, making it easier to identify the non-smooth contours of turbulent particles.

The unstructured 3D mesh in XGC1 is described in the `mesh` file, which is static over time, and specifies the points and connectivity of a single plane, and the number of planes around the tokamak. Medium and large runs will use about 100K and 1M points per plane respectively.

The `output.bfield` file contains the steady state magnetic field defined on the unstructured mesh and is static.

The `oneddiag` file contains general diagnostics that are appended after each time step. This file contains around 80 different diagnostic values, such as densities, flow, and momentum values, and is used to calculate a number of derived quantities.

The `3d` file is produced every time step and contains data for each plane in the simulation. The data is partitioned based on the underlying triangular mesh describing the tokamak. That is, this data is produced during the deposition and data reduction step in the simulation, where raw particle data is deposited onto the triangular mesh, producing an average value for that mesh region.

The `f3d` file is produced every time step and consists of ion and electron information relating to temperature, density, and velocity. The data is partitioned just as in the `3d` case, and is based on the underlying triangular mesh describing the tokamak, resulting in an average value for each mesh region.

Table 5 contains a summary of the previously detailed information on XGC1 output files and associated file size.

Table 5. A summary of the output data from XGC1 that is used most often by those interviewed. The table shows average sizes for medium and large runs, as well as how often the data changes.

	File Size (GB)		
File Name	**Medium Run**	**Large Run**	**Output Frequency**
restart	976	19,531	1-100 Time Steps
restartf0	48	522	1-100 Time Steps
mesh	0.025	0.256	Static
output.bfield	0.075	0.75	Static
oneddiag	0.002	0.03	Every Time Step
3d	0.075	0.8	Every Time Step
f3d	0.35	2.0	Every Time Step

3.4 XGC1 User Surveys

The XGC1 project is composed of a large membership, including physicists, experimentalists, analysts, and computer scientists. This diversity of backgrounds leads to a broad range of activities to be performed on various parts of the data, each requiring varying computational and data resources. In order to gain a holistic understanding of the project, we conducted interviews with 7 different XGC1 team members, covering key areas of the XGC1 workflow. Our interviews started with the same questions for each participant, although follow-on questions were adapted based on the interests and expertise of the participant. From these interviews we have distilled a list of required and "nice-to-have" analysis routines on XGC1 data. Finally, while our interest in these requirements is in how they apply to in situ processing, we note that in many cases they are applicable to post hoc processing requirements as well.

The required and nice-to-have analysis routines can generally be categorized into three areas: (1) visualization and analysis, (2) simulation monitoring, and (3) debugging and performance engineering. For each of these three areas we will report on our findings from our interviews, as well as indicate which of the items

is a Data Analysis and Visualization (DAV) task. DAV's are specific instances of the requirements we identified through our interview process. One key finding from the interviews, which is highly relevant for in situ, is that XGC1 allows up to 10% of total simulation time to be devoted to I/O. This fact must be kept in mind as new data requirements and fidelities are output for visualization and analysis tasks. The requirements gathered from the XGC1 team in each of the three areas are presented in Sections 3.4.1, 3.4.2, and 3.4.3 respectively.

3.4.1 Visualization and Analysis. A common analysis task in XGC1 is to make an image of a feature or region of interest. Images can serve several distinct functions in XGC1: (1) a diagnostic tool for checking new physics in the code, (2) a debugging and verification mechanism for new visualization routines, and (3) a method of exploring, discovering, and understanding new properties in the tokamak that either were not known or have been assumed to exist by the physics community. There are two types of images needed from XGC1: static plots and videos of time varying quantities.

3.4.1.1 Make Static Plots. Static plots are images of particular regions or quantities in the simulation. These plots include graphs, contour, histograms, pseudocolor plots, etc. The following are commonly created plots:

- **DAV 1:** Plots of the scalar value potential over time. This requirement primarily draws data from the `3d` file.
- **DAV 2:** Plots of heat flux, turbulence, or the temperature on surfaces over time. This requirement primarily draws data from the `f3d` file.

– **DAV 3:** Plots of the moments of the distributions functions (first order, second order, third order) of the different XGC1 variables: density, kinetic energy, etc. This requirement primarily draws data from the `f3d` file.

3.4.1.2 Make Videos.

Videos show the evolution of the simulation over time. The most common types are field and particle videos. Field videos show the statistical properties of the particles on the mesh. Particle videos show particle evolution, requiring very large amounts of data due to the large number of particles. The plots from DAV 1, DAV 2, and DAV 3 can also be made into videos, but some common analysis tasks that only make sense when shown as an evolution over time include:

– **DAV 4:** Average vector in a region, as shown in Figure 8a. This video type primarily uses data stored in the `restart` and `mesh` files.

– **DAV 5:** Rendering particle paths as they progress around the tokamak. This video type primarily uses data stored in the `restart` file.

– **DAV 6:** Detecting and visualizing particles that collide with the tokamak wall, as shown in Figure 8b. A requirement of this DAV task is the identification of particles that collide with the wall at some point in the simulation. This requires two-passes over the data, one to identify the particles that collide with the wall at any time, and the second to render these identified particles and the collisions with the tokamak wall. After the collision, these particles are removed from the scene. Because of the large size of the particle data, and two passes over all time steps are required, there is no known way to perform this task in situ. Even running the simulation run twice (once to identify particles, and the second time to render identified

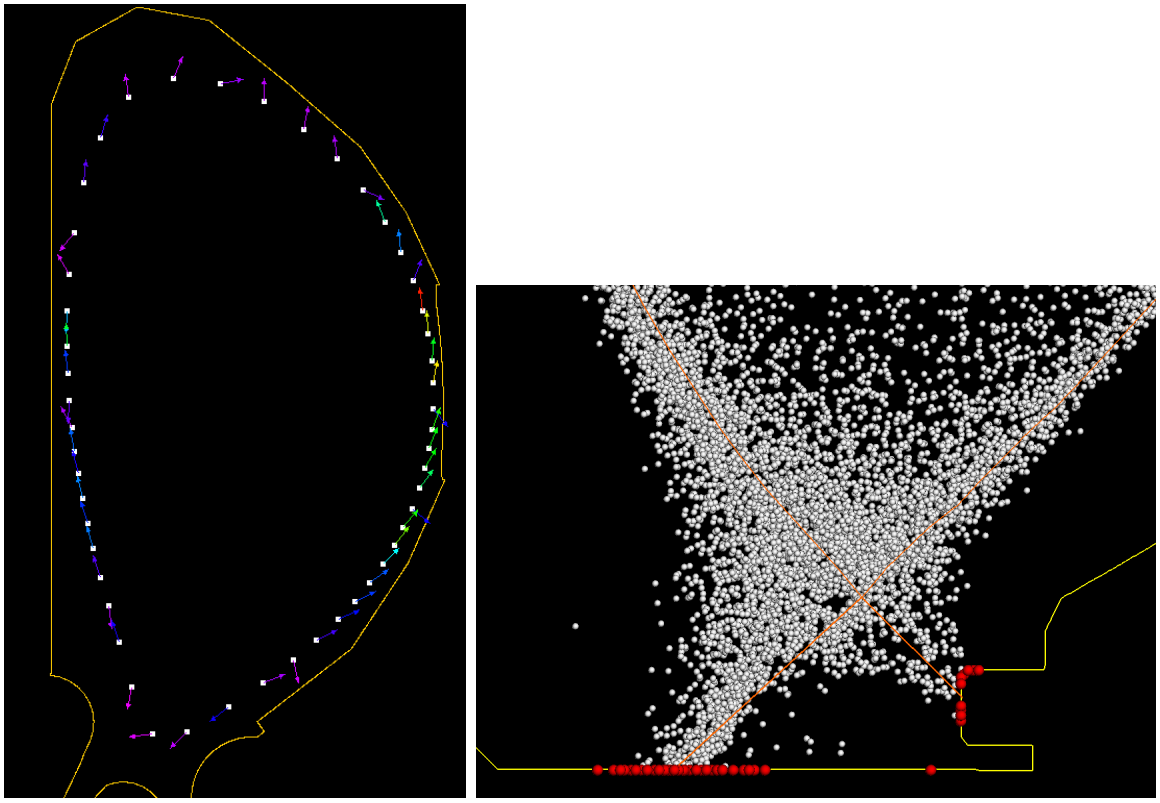
particles) can be problematic, since the particles are not guaranteed to be reproducible across runs. This video type primarily uses data stored in the `restart` and `mesh` files.

– **DAV 7:** Visualizing the turbulence derived quantity. This video type primarily uses data stored in the `3d`, `mesh`, and `oneddiag` files.

3.4.1.3 Interactive Visualization and Analysis. Interactive visualization and analysis is accomplished using ADIOS [82] and data staging, where data are streamed from the XGC1 simulation to a data server for visualization. The main interactive visualization task in XGC1 is blob tracking:

– **DAV 8:** Blob tracking involves identifying areas of high energy within the plasma which can form nonlinear turbulent eddies. The longevity, size, shape, and composition of these eddies are interesting to researchers, and their visualization gives insight into their 3D structure and perturbation to particle orbits. Blob tracking requires regions of interest to be identified through user interaction, and then the particles composing the blobs in those regions are tracked in subsequent time steps. This task is important because blobs represent areas of high energy and temperature which can damage the wall of the tokamak. Understanding the development and nature of blobs is crucial to the design and operation of tokamaks. The data used in this analysis includes data from the `restart`, `3d`, `mesh`, and `oneddiag` files.

3.4.1.4 Synthetic Diagnostics. Synthetic diagnostics provide a way to compare simulation and experimental data. Generally, experimental data are not directly comparable to the outputs of simulations, and so a transformation step is often required. Once transformed, experimental data can be used to verify



(a) Average PSI velocity in a region in XGC1

(b) Accumulation of particle impacts to the containment vessel wall

Figure 8. Example frames from XGC1 analysis videos demonstrating common visualization tasks.

simulation results. These capabilities are currently under development, so no measurable data analysis and visualization task exists yet for this requirement.

3.4.2 Simulation Monitoring. Simulation monitoring is concerned with real or near-real time reporting of simulation status to the scientists. This monitoring can include tasks such as creating plots of important variables or functions as the simulation progresses, detecting bad simulation states and halting the simulation, and even simulation steering by sending instructions from the monitoring routine back to the simulation.

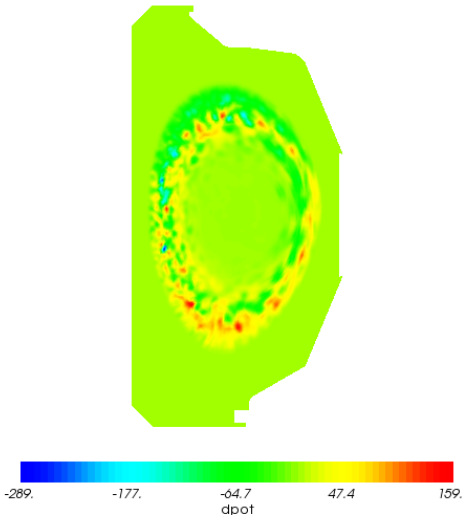
3.4.2.1 Simulation Dashboard.

A simulation dashboard is an easy to access web page from which scientists can remotely access key information about running simulations, as well as past simulations. For data that cannot be appended to existing plots at each time step, the dashboard must allow a mechanism to explore plots over time. It should enable support for continuing a past simulation run on the same dashboard, and contain links to the storage locations for the data used in each of the visualizations for each run, making retrieval of data related to interesting aspects of a run easy. The dashboard visualization requirements are as follows:

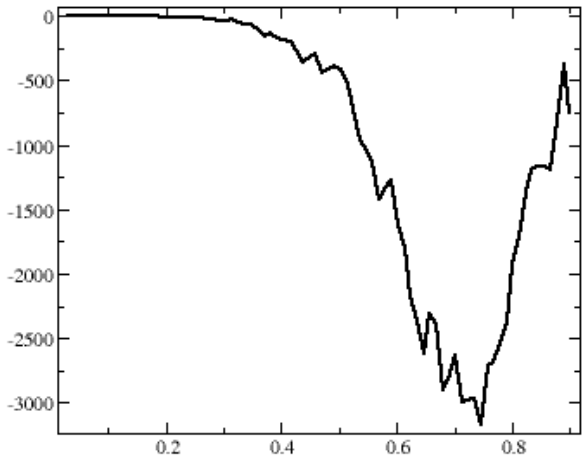
- **DAV 9:** Plotting values on each of the poloidal planes of the simulation for every time step, as shown in Figure 9a. The number of planes that are plotted are equal to the number of simulated poloidal planes in the tokamak, typically 16, 32, or 64, plus one plot that represents averages of the values of all planes. This requirement primarily draws data from the `3d` and `mesh` files.
- **DAV 10:** Plotting all of the variables contained in the `oneddiag` file for each time step, as shown in Figure 9b. Typically this produces 150 different plots.
- **DAV 11:** The automatic creation of a video summarizing each variable at the end of the simulation, a video of the average planes from DAV 9, and videos summarizing the slices of the torus.

3.4.3 Debugging and Performance Engineering.

There are a number of debugging and performance tasks that are desired, or in the works, for XGC1, but, at present, they are not part of the production codebase or analysis and visualization workflows. We therefore have no DAV tasks to report. However,



(a) Example of a slice plot of *potential* at one time step



(b) [Example of a variable plot showing *poloidal flow* over time

Figure 9. Example images produced by an XGC1 online dashboard during one simulation time step.

we include a discussion on the major items on the wish list to illustrate directions for future development.

3.4.3.1 Debugging. Debugging code related to the introduction of new physics or performance enhancements in XGC1 is always challenging. Worse, many problems only occur when running at very large scales.

- *Error Logs* are one method of debugging, and provide a great source of information, though is generally underutilized. The ability for analysis and visualization of these logs could provide useful feedback.

- *Particle Loss* is the loss of particles from the tokamak containment vessel, as shown in Figure 10. Recent particle loss has manifested near the simulation boundaries. This information is currently saved to the error log and retrieved after the run is over.

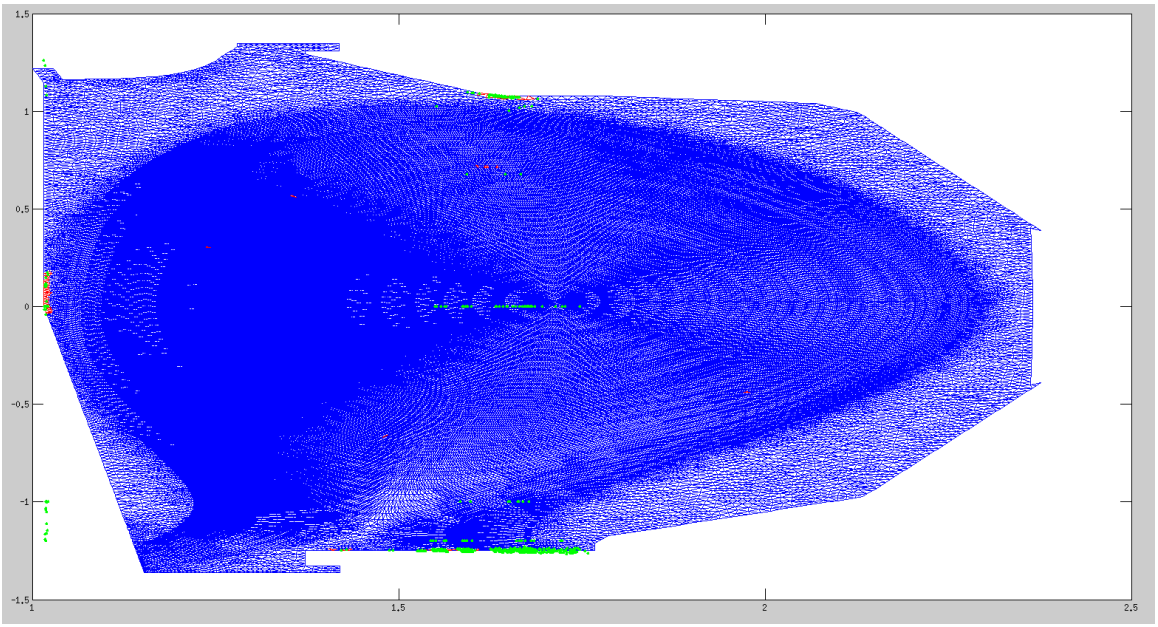


Figure 10. Debugging image that shows where particle loss was occurring in the tokomak containment vessel.

3.4.3.2 Low Level Monitoring. No low level monitoring exists in the XGC1 code, meaning that the code will not stop itself once the results become invalid. This is an opportunity for improvement. For example, checks to detect when a certain percent of particles have been lost from the simulation (making the results invalid) could be implemented.

3.4.3.3 Work Division (load balancing). Work division is the process of balancing the distribution of particles to processor ranks in a plane of the simulation. Three possibilities exists for balancing the particles in an XGC1 plane: (1) the toroidal direction, (2) the poloidal direction, or (3) a hybrid combination of the two. For context, the toroidal direction is the long way around the torus, and the poloidal direction is the short way around the torus.

– *Toroidal Load Balancing* is currently being done in production. Experiments indicate this method yields the biggest performance gains.

– *Hybrid Load Balancing* is under experimental development. At this time it is not clear if this type of load balancing would benefit the overall runtime of the simulation. This is due to the fact that poloidal motion is very fast and intuition tells them that it does not end up being a problem. However, further studies into this could be beneficial.

– *Imbalance detection:* XGC1 currently has no mechanisms for detecting when particle imbalance begins to become a detriment to performance, and when a rebalance would be worth the overhead cost. Further work and analysis would prove useful.

3.4.3.4 Collision Detection.

Collision detection is a feature under development, and attempts to balance the simulation by collisions between particles (currently only a single species, but multiple species would be useful). Methods are wanted to visually compare load imbalances by collision versus particle imbalances to answer the question of how these imbalances are different, and how to optimize for both.

3.5 Summary

We surveyed a diverse set of people associated with the large-scale fusion simulation code XGC1, gained an understanding of how they work, and cataloged their visualization and analysis requirements for in situ processing. This look at the breadth and depth of in situ requirements for a large-scale simulation code provides valuable insight into the needs of a diverse team. The identified DAV's vary drastically in terms of computational and data resources required, demonstrating a wide breadth of needed in situ flexibility and capability. Finally, we believe the breadth of requirements for XGC1 will be similar for other simulations codes, but

that a study such as ours would need to be repeated for these teams to gain an in depth understanding.

This study suggests several interesting directors for future work. First, there is a need for a classification scheme in order to evaluate in situ tasks. That is, some tasks may be best suited to run in-line, while others may be best suited to run in-transit, and some even may be best as post process tasks. In Chapter IV we present a work that identifies evaluation factors for evaluating the efficacy of a task for post hoc, in-line, or in-transit implementation Second, there are large differences between the data and computational requirements of many of the XGC1 visualization and analysis tasks. These differences will lead to variations in compute and time resources that need to be dedicated to each task. In addition, these differences will likely vary for a given task depending on the timeliness of the result needed as well as the scale that the simulation is being run. In Part II we present two studies conducted to evaluate different classes of visualization algorithms at varying scale in order to understand their scaling and timeliness curves. Those studies will aid in enabling simulation scientists to make the most efficient use of their time and compute resources.

CHAPTER IV

COMPARISON FACTORS FOR EVALUATING IN-LINE AND IN-TRANSIT IN SITU

Most of the text in this chapter comes from [70], which was a collaboration between Scott Klasky (ORNL), Norbert Podhorszki (ORNL), Jong Choi (ORNL), Hank Childs (UO, LBL), David Pugmire (ORNL), and myself. The writing of this paper was a collaboration between Hank Childs, David Pugmire, and myself, and I performed the lead role on all writing. Hank Childs, David Pugmire, and I primarily created and classified all of the comparison factors in this work. Scott Klasky, Norbert Podhorszki, and Jong Choi were involved in initial discussions and provided edits to the manuscript.

In this chapter, we explore a set of factors by which in situ paradigms can be evaluated and ranked for a given application scenario. The ten comparison factors that we present span a range of issues relevant to both scientists that are running simulations, and computer science researchers and developers that are developing analysis and visualization methods. The purpose of these factors is to give researchers a starting point for evaluating which in situ paradigm will be the most effective for their given circumstances. Throughout this chapter we present our recommendation on which in situ paradigm will likely benefit the most for a given comparison factor, and maintain that in-transit in situ will play an important role for in situ workflows for the foreseeable future. We start by motivating the need for in-transit in situ and a set of comparison factors, describe the comparison factors, discuss the interplay between the factors, and then show a subset of the factors in practice in a scientific workflow.

4.1 Motivation

As discussed in Chapter I, there are two major paradigms for in situ processing, and it is unclear which paradigm simulation code groups and visualization software developers should back, and under what circumstances. To address this, this chapter presents 10 factors for objectively comparing an in situ visualization approach in a given circumstance.

The remainder of this chapter is organized as follows: Section 4.2 compares and contrasts both paradigms against the following set of 10 factors: data access, data movement, data duplication, data translation, coordination, resource requirements, exploratory visualization, scalability, fault tolerance, and ease of use. Section 4.3 presents our perspective for why in-transit in situ visualization is an important technique to consider for future and current work in in situ. Section 4.4 provides a motivating use case of in-transit in situ demonstrating a subset of the comparison factors in practice. Section 4.5 presents our final thoughts on the long-term benefits of in-transit in situ.

4.2 In Situ Comparison Factors

The comparison factors selected were intended to span the range of issues relevant to both scientists that are running simulations, and computer science researchers and developers that are deploying analysis and visualization methods. These factors consider required HPC resources (both shared and dedicated), impact on the running simulation, fault tolerance, and usability.

4.2.1 Data Access. With simulations producing more data than can be saved to disk, a different data set is available for visualization and analysis depending on when the data are accessed (in-line, in-transit, or from file). Generally speaking, there are more data and time steps available on the simulation

resources than there will be once the data are transferred and saved to disk. This makes it important that the correct set of operations are performed on the data at each stage. For operations that require all data and all time steps, that operation should be performed on the simulation nodes before data are culled. However, if an operation or simulation team can handle performing analysis on a sparser data set, that operation could take place after data are saved to disk.

With in-line in situ, visualization and analysis routines can take advantage of having the full richness of the simulation output. Operations can be done that take into account all of the produced data for every time step.

In-transit in situ visualization routines on the other hand, often must operate with a sparser set of data. However, it should be noted that this data set can be more complete than those that are saved to disk, because the network transfer can allow for a greater volume of data to be sent. Therefore, in-transit in situ routines often work with less data than is available in situ, but more than is available post hoc.

Favored Paradigm: in-line in situ

4.2.2 Data Movement. Moving large quantities of data from one location to another can be an expensive task. The cost of this task varies substantially depending on where the data are being sent, i.e. between nodes in an allocation or off over the network, so data movement should be kept to a minimum.

Often the amount of data needed varies by the visualization algorithm employed. For a simulation using in-line in situ visualization and analysis, the amount of data moved can range from none, to simulation stalling levels. This is because some visualization algorithms traditionally require large amounts of data to be sent between the ranks, which complicates the problem when using in-line

in situ. Communicating between every node in the simulation can be enormously expensive compared to a smaller node allocation.

In-transit in situ visualization has a different issue with regards to data movement. Before in-transit in situ visualization can take place, the data must be sent from the simulation to a visualization resource for processing. This dump from the simulation to the visualization resource can saturate the network, and could even cause a slowdown in the simulation while it sends the data off over the network. This data dump though has the potential to end up moving far less data, in total, during the visualization routine vs. that of in-line in situ. This is due to visualization allocations traditionally being much smaller than simulation node allocations, meaning that communication takes place over a much smaller domain.

Favored Paradigm: draw

4.2.3 Data Duplication. At the conclusion of each time step of a simulation, a new set of data are available and ready for use. On node resources may take immediate advantage of this data, while off node resources require a copy to be made. The act of making this copy means that the data now exists in two places, doubling the memory footprint.

In-line in situ visualization does not have a data duplication problem. All data are already available within the simulation, so no duplication will take place.

In-transit in situ visualization must work on a copy of the data by definition. That is, the data are copied from the simulation nodes to whatever in-transit in situ visualization solution is being used. This duplication now doubles the RAM usage for each time step, possibly making it the less efficient choice.

Favored Paradigm: in-line in situ

4.2.4 Data Translation. Simulation codes store mesh and field data in myriad ways that visualization programs must be able to interpret and work with. The foundation for performing such a translation is a data model (which describes what data can be represented) and its implementation (which describes how to lay out arrays).

In the in situ world, there are two basic options. First, the visualization code can allocate new arrays that match its own data model implementation and then copy data from the simulation code's arrays into its own arrays. Obviously, this memory bloat is often viewed as undesirable. However, this approach is still used in VisIt's LibSim and ParaView's Catalyst. The second option is to ensure that the visualization code can work on directly on the simulation data layout. This is straightforward when writing custom code specifically for that simulation, but much harder when trying to design a general purpose visualization infrastructure that can be re-used with many simulation codes. The approaches used by the community so far involve redirection of data accesses through virtual functions (done in some cases with Catalyst), designing a data model implementation that support many different array organizations to increase the chances that the simulation code uses an array layout that the visualization code can support (as with EAVL), or writing templated code that is customized to the simulation code during the compilation process (as with SciRun).

To date, the two basic options have proven to be difficult for doing easy and overhead-free data translation. Instead, we note that this problem has been addressed previously, for data I/O, where simulation codes write arrays to disk and visualization codes read them. Establishing schemas, interfaces, and conventions was a non-trivial task in this space, but one that is now generally considered

"solved." With respect to in situ, the in-transit approach can take advantage of this existing solution, by using the simulation code's I/O calls as a way to pass data. As a result, the path to integrating in situ technology with the loosely coupled approach is significantly less of a burden.

Favored Paradigm: in-transit in situ

4.2.5 Coordination. Coordination is required between the simulation and the visualization. This coordination lets the visualization know that the next iteration of simulation data are ready and that visualization can begin.

In a in-line in situ paradigm coordination is minimal. If visualization code is directly embedded into the simulation, this could be as simple as calling the visualization routine at the end of the simulation main loop. For production tools like LibSim and Catalyst the coordination is very similar, but the call is made into the particular library.

In a in-transit in situ paradigm much more coordination is required. At the end of each cycle in the main loop a call must be made to transfer the data to the visualization resource. This transfer requires use of the network and coordination on both the sending and receiving side to ensure the data are successfully sent and received. To guard against faults, care must be taken to recover from situations when a network call fails, or the visualization resource is not available.

Favored Paradigm: in-line in situ

4.2.6 Resource Requirements. All in situ paradigms require additional resources of some sort. In a in-line in situ paradigm the simulation and visualization share the same resources, including execution, memory, and network. In an era when memory per core is steadily decreasing, visualization tools are required to operate under very tight memory restrictions. In cases

where intermediate results need to be computed and held in memory, this can be a challenge. Additionally, super computing time is in high demand, and very expensive. Therefore simulations will generally dedicate a fixed window of time for visualization. These restrictions place challenges on visualization which have generally run on dedicated resources with large memory, or on the development of new techniques that operate within tight time and memory requirements.

In a in-transit in situ paradigm additional visualization nodes are required. These additionally nodes are requested at the time the simulation is run, add to the cost of running a simulation. However, these additional nodes can be used asynchronously once the data are transferred. The visualization can run while the next time step is being computed by the simulation, and there are no restrictions on memory usage. However, care must be taken to handle the arrival of the next time step if the visualization routines are still running. But otherwise, the restrictions are minimal.

Favored Paradigm: draw

4.2.7 Exploratory Visualization. Exploratory visualization, a task most associated with post processing of data on disk, is generally, not a strength in any in situ paradigms. Typically, the visualization that is done must be specified a priori, and so care must be taken to decide when the simulation is launched which particular operations will be performed. However, tools like LibSim and Catalyst do allow fully featured visualization tools access to specified parts of simulation data, making free-form exploratory visualization possible, but at the expense of pausing the simulation while the user interacts the data.

Favored Paradigm: draw

4.2.8 Scalability. Any in situ paradigm is constrained to use the concurrency of the allocated resource. In a in-line paradigm this is the allocation for the entire simulation. While this level of concurrency might be advantageous for embarrassingly parallel routines that require little synchronization or communication, it can be a bottleneck for visualization routines that require significant communication (e.g. particle tracking, etc), or algorithms that don't exhibit scaling up to the levels of simulation codes (e.g. hundreds of thousands of cores). Conversely, in a in-transit paradigm, the concurrency of the visualization resource can be appropriately configured for the tasks to be performed. Algorithms that require significant synchronization and communication will generally perform much better at lower levels of concurrency, and this can be used to optimize the performance.

Favored Paradigm: in-transit in situ

4.2.9 Fault Tolerance. As supercomputers continue to grow in size and complexity, resilience and fault tolerance at all levels become increasingly important. For in-line in situ paradigms, where visualization and simulation run together, fault tolerance becomes imperative. Simulations are directly exposed to data corruption, infinite loops, or errors in visualization routines, and could result in faults or crashes. Because of the expense of super computing time, and the drastic impact of faults on simulation codes, fault tolerance is a requirement. Something that in practice is very hard to achieve.

Because of the clear and distinct separation between the simulation and the visualization in a in-transit paradigm, the exposure to faults is greatly reduced. In this paradigm the data transfer to the visualization resource becomes the

only point of exposure to faults. The exposure can be further reduced by using asynchronous transfers.

Favored Paradigm: in-transit in situ

4.2.10 Ease of Use. Usability spans a wide range of topics, and includes things such as integration, deployment, development, and dependencies. For in-line in situ, where there is a fundamental connection between the simulation and visualization code, software engineering practices become very important. Because of this basic interdependence, changes in either the simulation or visualization code, or dependencies on third party libraries need to be carefully managed. In the case of stand-alone production packages where there is a more separated interface point, careful coordination of releases and patches is still required.

For in-transit in situ the interface between the simulation and visualization takes place through the API. Here, a cleanly defined, concise and small set of APIs determine the usability of the system.

Finally, there is no free lunch. Development costs must be taken into account. While writing custom visualization code has the advantage of maintaining full control and making domain-specific optimizations easy, there is the cost of not taking advantage of community-wide investments devoted to making standard tools and libraries. On the other hand, developing in-transit in situ frameworks is a large undertaking, and providing the flexibility to handle a wide variety of uses cases is a challenge.

However, given the advantages afforded by the separation of simulation and visualization, the in-transit paradigm occupies a much stronger position.

Favored Paradigm: in-transit in situ

4.3 Discussion

Based on the evaluation of the 10 factors we considered, there are clearly very good reasons for using different techniques. In cases with very specific needs, there is often a clear choice. In practice however, there are generally many factors under consideration, and we hold that some factors are much more important than others. In particular, we hold that fault tolerance, ease of use, and data translation are the most important of the 10 factors discussed.

As discussed in Section 4.2.9 the increasing complexity of supercomputers and the workflows being run on them makes fault tolerance of paramount importance. The ability of in-transit in situ to completely separate the simulation from the visualization makes it the clear choice.

On a related note, the complete separation of simulation and visualization in a in-transit paradigm is a large contributor to the win for ease of use concerns (see Section 4.2.10). This minimization of contact points between the two, along with the flexibility provided with configuration of simulation runs and setup of visualization choices make in-transit in situ the clear choice.

Finally, as discussed in Section 4.2.4, the diversity of data models and data layouts in simulation codes makes efficient interfacing of simulation outputs and visualization a daunting challenge. In-transit in situ methods solve this problem by doing what simulations and visualization routines already do, writing and reading data. Simulations do not even have to be aware of what happens after data transfer calls are made, the underlying system takes care of transferring the data, and the visualization access the data by making data read calls.

The advantages of in-transit in situ in these key areas makes it clear that this paradigm should be a staple in visualization now, and going forward. As

a testament to the viability of this paradigm, in-transit techniques have been demonstrated with production runs on some of the largest super computers in the world [22, 116, 44].

Finally, there is one final and critical point for consideration. Hybrid methods [38], where both in-line and in-transit paradigms are used at the same time, are an exciting and very promising direction. These methods support the flexibility of processing data on the simulation resource before they are either written to disk, or transferred to the visualization resource for additional processing. In other words, it offers the ability to achieve the best of both paradigms. However, hybrid methods are *only* possible within a context that supports in-transit in situ. It is otherwise impossible.

4.4 Considering these Factors with an XGC1 Integration

The setup we employ places an emphasis on several of the factors discussed in Section 4.2, including ease of use, fault tolerance, data translation, scalability, and resource requirements. This maps most directly onto a in-transit in situ paradigm. Our workflow consists of three primary elements: (1) the simulation code; (2) a data transfer system to move data from the simulation to the visualization nodes; and (3) an efficient parallel visualization library. The simulation code, XGC1 [33], is a highly scalable physics code used study plasmas in fusion tokamak devices. For the latter two elements, we utilize three important libraries which are described below: ADIOS and DataSpaces for data management and transfer, and VTK-m as a framework for light weight visualization plugins.

The in-transit paradigm in ADIOS and DataSpaces provides for a clean interface and separation from XGC1 that provides ease of use, and fault tolerance. The ability to control the concurrency of the visualization tasks independent of the

concurrency of XGC1 is important for ensuring good scalability on the visualization nodes. Further, the resource requirements can be specified based on the types of visualization that will be performed. The VTK-m framework offers a data model with the flexibility to efficiently, and optimally represent the output format for XGC1.

4.4.1 ADIOS. The Adaptable I/O System (ADIOS) [82], is a componentization of the I/O layer that is accessible via a posix-style interface. The ADIOS API abstracts the operation away from implementation, allowing users to compose their applications independent of the underlying software and hardware. This capability, along with the functionality of DataSpaces [43] allows this same API to support read/write operations from/to the memory space of visualization nodes.

This type of in-transit in situ provides significant advantage for one of the most important factors considered, namely ease of use. It is worth emphasizing that in-transit in situ is achieved with minimal modifications to the simulation code. It uses something the simulation is already doing, namely I/O. These further address two of the most important factors, ease of use and fault tolerance.

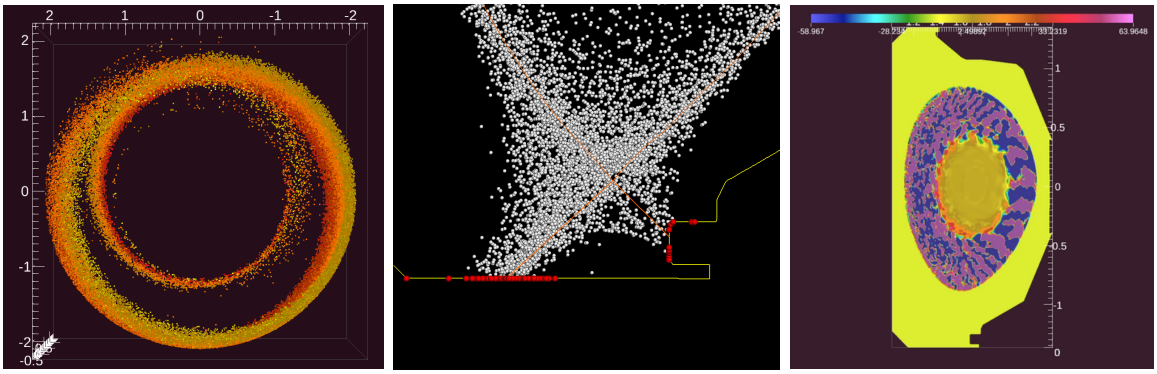
4.4.2 Visualization Plugins. We designed our visualization routines as flexible, light weight plugins. Our plugins are based on an emerging community standard, VTK-m [123], which is a project building upon the success of three existing visualization frameworks, Dax [97], PISTON [83], and EAVL [93, 94]. The VTK-m framework is targeted to emerging computational systems where parallelism and the use of accelerators are dramatically increasing, and memory per core is decreasing. An emphasis has been placed on much more powerful data

models that allow efficiencies in representing the various mesh types and data layouts used by simulation codes.

4.4.3 Visualization Workflows for XGC1. In previous work we utilized the features of ADIOS and EAVL (as a precursor to VTK-m), and demonstrated the effectiveness of in-transit in situ visualization for large scale simulation codes using a workflow consisting ADIOS, data staging and EAVL [116]. In that work we focused on the performance, scalability, and ease of use of visualization plugins that were used on the output of the XGC1 simulation code.

In that study we performed visualization on two different output fields from XGC1, the plasma particles (both ions and electrons), and field variables from the unstructured mesh. The ease of use of this system was highlighted with the fact that no changes to XGC1 were required. All modifications to data movement were accomplished with only a change to the ADIOS configuration file. At each simulation step, particles of interest were identified and visualized (Figures 11a and 11b) in parallel along with the visualization of a slice plane through the mesh, allowing us to monitor simulation field data, such as plasma turbulence (Figure 11c). These images were then used for monitoring the simulation and for post run analysis.

Using the factors from Section 4.2 to compare the two paradigms highlights the advantages of a in-transit paradigm. Using the the ADIOS API, no modifications are made to the simulation code to send data to the visualization nodes via DataSpaces. The only change required is to the ADIOS configuration file which is read when the simulation starts. This affords large advantages in both ease of use, and fault tolerance. Further, the data translation issues are avoided since the simulation code writes data in a known format to ADIOS, which flows



(a) Selection of particles of interest.

(b) Particle interaction with vessel boundary.

(c) Slice plane of field data.

Figure 11. Representative examples of XGC1 particle and field visualization performed with a in-transit in situ paradigm.

to the visualization nodes, and is then read by the visualization plugin. This also highlights the ease of use advantage since the visualization is doing something that it already does, namely, read data. Further, the separation of simulation and visualization resources further highlights ease of use by eliminating any dependencies between the simulation and visualization code, as well as providing a layer of protection through fault tolerance.

4.5 Summary

In summary, this chapter presented 10 factors for comparing in-line and in-transit in situ paradigms. Based on our evaluation of these 10 factors, there are clearly very good reasons for using each of the techniques. Table 1 summarizes the evaluation metrics, and which in situ paradigm is best in that area. In cases with very specific needs, there is often a clear choice of in situ method. In practice however, there are generally many factors under consideration, and the optimal in situ approach will be situationally dependent.

In-line approaches will work very well when the simulation has a predefined list of images and analyses that it needs produced. These can be directly coded

into the simulation. We emphasize that if a visualization task has low inter process communication and can be in-line, it is generally best to do so.

On the other hand, if interactive exploration is required, if subsets of data should be saved for further analysis, or an open source visualization solution needs to be employed for data visualization, in-transit might be the best approach. These approaches avoid many of the pitfalls of being fault intolerant, they are generally easier to deploy and maintain, they generally scale better, and the data translation from the simulation representation to the visualization library representation is solved through existing I/O calls. And finally, we note that the significant advantages to be gained using a hybrid paradigm can *only* be realized within a system that is based on a in-transit paradigm.

Since both paradigms are strong under varying circumstances, further study is needed to model visualization algorithms from small to large scale in order to obtain a performance profile that can be used for deciding how to place in situ visualization tasks. In Part II of this dissertation, we focus heavily on the *Scalability* evaluation metric, and gather performance profiles for two common visualization algorithms to understand how time to solution and overall cost vary between in-line and in-transit methods.

Part II

Findings

In this part of the dissertation, we explore and answer the questions introduced in Section 1.2. We begin by describing the infrastructure and tests that we conducted, and give an overview of the data we gathered (Chapter V). We follow this introduction with in-depth explorations of both the time (Chapter VI) and cost (Chapter VII) of in situ techniques. Finally, we conclude with answers to each of the questions posed in this dissertation, and list multiple areas of interesting and valuable future work (Chapter VIII).

CHAPTER V

CORPUS OF DATA

Most of the text in this chapter comes from [71] and [73], which were collaborations between Scott Klasky (ORNL), David Pugmire (ORNL), Matthew Wolf (ORNL), Norbert Podhorszki (ORNL), Jong Choi (ORNL), Mark Kim (ORNL), Matthew Larsen (LLNL), Hank Childs (UO), and myself. The experimental infrastructure for [71] was primarily constructed by myself. Matthew Larsen consulted on functionality and use of some of the infrastructure components. I ran all the experiments, compiled the results, created all of the initial analysis and figures, and wrote the majority of the manuscript text. David Pugmire was involved in helping to design the experiments, providing help and instruction for getting all of the tests completed, as well as editing of the final manuscript. Hank Childs provided extensive feedback during the work and was involved in editing the manuscript. Scott Klasky, Matthew Wolf, Norbert Podhorszki, Mark Kim, Jong Choi and Matthew Larsen were involved in initial discussions and provided comments on the final manuscript. The experimental infrastructure for [73] was primarily constructed by myself. Matthew Larsen consulted on functionality and use of some of the infrastructure components. I ran all the experiments, compiled the results, created all of the initial analysis and figures, and did a significant amount of writing for the manuscript. David Pugmire was involved in helping to design the experiments, providing help and instruction for getting all of the tests completed, as well as being heavily involved in editing of the final manuscript. Hank Childs provided extensive feedback during the work and had a major hand in writing sections of the report. Scott Klasky, Matthew Wolf, Norbert Podhorszki,

Mark Kim, Jong Choi and Matthew Larsen were involved in initial discussions and provided comments on the final manuscript.

In this chapter we introduce our corpus of data from which we draw our findings for the following two chapters. We introduce our software, hardware, run configurations, and perform an overview of all of the data that we collected. This chapter serves to reduce redundancy in Chapter VI and Chapter VII, as such, we will not reintroduce this information, but will instead refer back to this one.

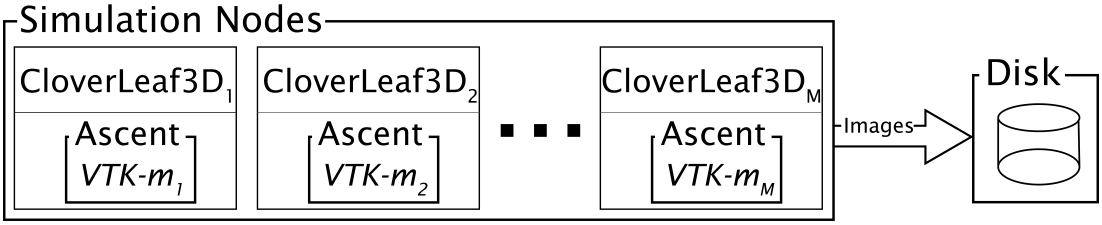
5.1 The Corpus

In this section we detail the experimental setup, methods, and software used to generate our corpus of data, as well as a cursory overview of the data we collected.

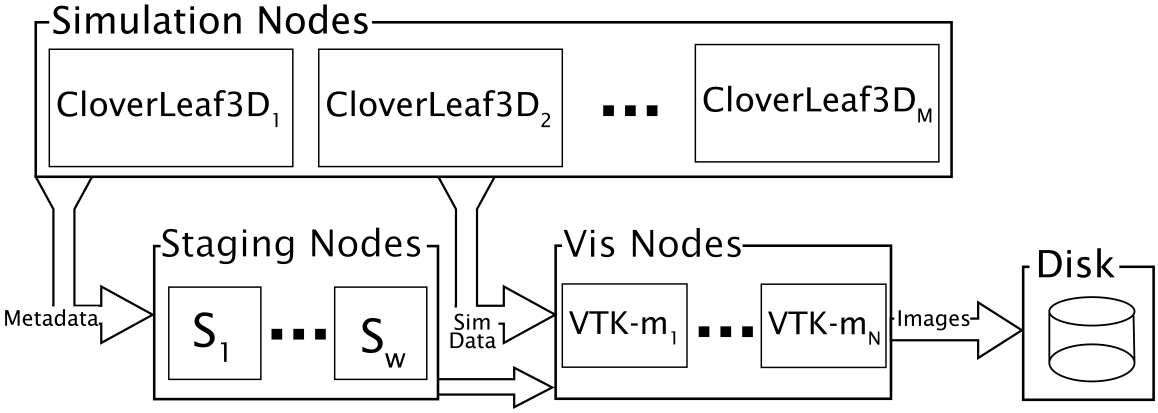
5.1.1 Experiment Software Used. To generate data for this study, we use CloverLeaf3D [2, 91], a hydrodynamics proxy-application. Cloverleaf3D spatially decomposes the data uniformly across distributed memory processes, where each process computes a spatial subset of the problem domain. To couple CloverLeaf3D with both in-transit and in-line in situ, we leveraged existing integrations with Ascent [77].

In-line visualization used Ascent's integration with VTK-m [103] for visualization operations. The visualization is described through a set of actions which Ascent turns into a data flow graph, and then executed. Figure 12a depicts how the software components interact in the in-line workflow.

In-transit visualization used Ascent's integration with the Adaptable I/O System (ADIOS) [82] to transport data from the simulation nodes to the in-transit nodes using its RDMA capabilities [43, 147]. ADIOS requires the use of dedicated staging nodes to hold the metadata necessary to service RDMA requests. Once



(a) In-line visualization setup. The simulation and visualization alternate in execution, sharing the same resources.



(b) In-transit visualization setup. The simulation and visualization operate asynchronously, and each have their own dedicated resources..

Figure 12. Comparison of the two workflow types used in this study.

the data are transported, the visualization tasks are performed using VTK-m. To be clear, the same VTK-m code was being used for both in-line and in-transit visualization. The only differences are the number of nodes used for visualization and the use of ADIOS for data transport to a separate allocation. Figure 12b depicts how the software components interact in the in-transit workflow.

5.1.2 Visualization Tasks Studied. There were two classes of visualization tasks in this study, computation heavy and one that is communication heavy. The computation heavy task was isocontouring and parallel rendering, while the communication heavy task was volume rendering. Visualization was performed after each simulation step. The computation heavy task consisted of creating two isocontours at values of 33% and 67% between the minimum and

maximum value of the simulation's *energy* variable, followed by ray tracing-based rendering. The ray tracing algorithm first locally rendered the data it contained, then all of the locally rendered images were composited using Radix-k [65]. The communication heavy task consisted of volume rendering the simulation's *energy* variable. Compositing for volume rendering is implemented as a direct send.

5.1.3 Application/Visualization Configurations. In this study we used five different in situ configurations of the application and visualization:

- **Sim only:** Baseline simulation time with no visualization
- **In-line:** Simulation time with in-line visualization
- **Alloc(12%):** In-transit uses an additional 12% of simulation resources
- **Alloc(25%):** In-transit uses an additional 25% of simulation resources
- **Alloc(50%):** In-transit uses an additional 50% of simulation resources

For in-transit visualization, pre-determined percentages of simulation resources for visualization were selected. These percentages were selected based off of a rule of thumb where simulation's typically allow up to 10% of resources for visualization. 10% was our starting point, and we then selected two additional higher allocations to explore a range of options. We initially considered in-transit allocations that were below 10%, but due to the memory limitations on Titan (32 GB per node), the visualization nodes ran out of memory. We leave a lower percentage study as future work on a future machine. Finally, we ran each one of these configurations with weak scaling with concurrency ranging between 128 and 32,768 processes, with 128^3 cells per process (268M cells to 68B cells).

CloverLeaf3D uses a simplified physics model. As such, it has a relatively fast cycle time. This fast cycle time is representative for some types of simulation's,

but we also wanted to study the implications with simulation's that have longer cycle times. We simulated longer cycle times by configuring CloverLeaf3D to pause after each cycle completes, using a sleep command. This command was placed after the simulation computation, and before any visualization calls were made. We used three different levels of delay:

- **Delay(0):** simulation ran with no sleep command.
- **Delay(10):** a 10 second sleep was called after each simulation step.
- **Delay(20):** a 20 second sleep was called after each simulation step.

Lastly, we ran each test for 100 time steps using a fixed visualization frequency of once every time step. This frequency ensures that fast evolving structures in simulation data are not missed. Also, very frequent visualization gives us an upper bound for how visualization will impact the simulation.

Due to the scheduling system on Titan each of our tests at different levels of delay had to be run on different days and times. These differences meant that the system load on other parts of the system varied from run to run, as did the location of our physical allocation on Titan. We were also not able to run each of the tests multiple times due to limited core hours being available on Titan. As such, the results that focus on total time or total cost are prone to variations due to noise, the most apparent variations happen when comparing across the different delay levels.

5.1.4 Study Hardware. All runs in this study were performed on the Titan supercomputer deployed at the Oak Ridge Leadership Compute Facility (OLCF). Because CloverLeaf3D runs on CPUs only, we restricted this study to simulation's and visualizations run entirely on the CPU.

Table 6. Resource configuration for each experiment in our scaling study.

Test Configurat	Sim Procs	128	256	512	1024	2048	4096	8192	16384	32768
	Data Cells	648^3	816^3	1024^3	1296^3	1632^3	2048^3	2592^3	3264^3	4096^3
In-line	**Total Nodes**	**8**	**16**	**32**	**64**	**128**	**256**	**512**	**1024**	**2048**
In-transit	Vis Nodes	1	2	4	8	16	32	54	128	256
	Staging Nodes	1	2	2	4	4	8	8	16	16
Alloc(12%)	**Total Nodes**	**10**	**20**	**38**	**76**	**148**	**296**	**584**	**1168**	**2320**
In-transit	Vis Nodes	2	4	8	16	32	64	128	256	512
	Staging Nodes	1	2	2	4	4	8	8	16	16
Alloc(25%)	**Total Nodes**	**11**	**22**	**42**	**84**	**164**	**328**	**648**	**1296**	**2576**
In-transit	Vis Nodes	4	8	16	32	64	128	256	512	1024
	Staging Nodes	1	2	2	4	4	8	8	16	16
Alloc(50%)	**Total Nodes**	**13**	**26**	**50**	**100**	**196**	**392**	**776**	**1552**	**3088**

5.1.5 Launch Configurations. The configuration for each experiment performed is shown in Table 6. Isosurfacing plus rendering was run on up to 16K cores, volume rendering was run on up to 32K cores. Because CloverLeaf3D is not an OpenMP code, the in-line in situ and the simulation only configurations were launched with 16 ranks per node. The in-transit configurations used 4 ranks per visualization node and 4 OpenMP threads to process data blocks in parallel. Therefore, in-transit and in-line both used 16 cores per node. Additionally, the in-transit configuration required the use of dedicated staging nodes to gather the metadata from the simulation in order to perform RDMA memory transfers from the simulation resource to the visualization resource. These additional resources are accounted for in Table 6 and are used in the calculation of all in-transit results.

5.1.6 Overview of data collected. In total, we ran 255 individual tests, each for 100 time steps. From each of these tests we collected the total time for each time step from both the application and visualization resources, as well as more fine grained timers placed around major operations. After the runs were

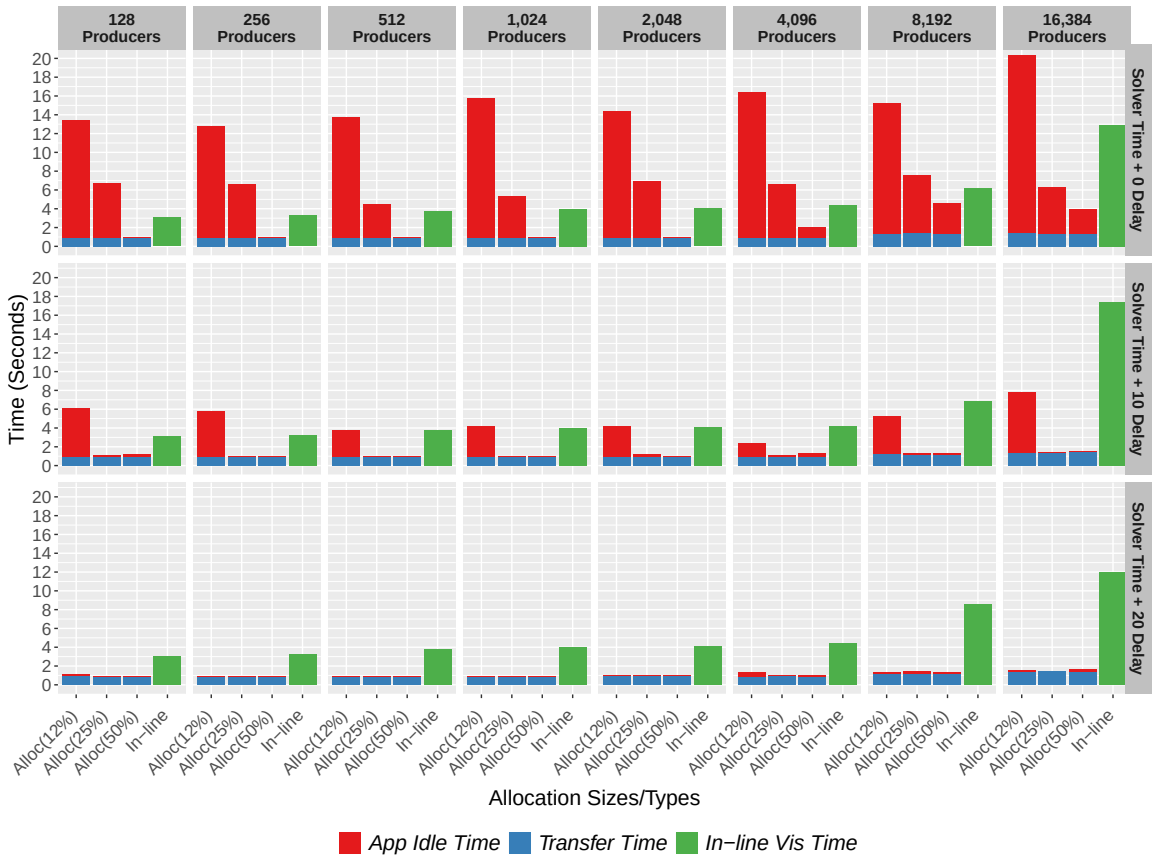


Figure 13. Stacked bar chart comparing the total time per step for using in-transit and in-line methods for isosurfacing plus rendering. This charts look at time from the applications perspective, meaning that the time for in-transit visualization is only how long it takes to transfer the data from the application, unless the in-transit resources block, in which case the application becomes idle. In-transit visualization is broken down into the time it takes to receive data data from the application and how long the application is blocked by the in-transit resources being too slow. In-line has a single time, how long it took to perform the visualization task. The application time is excluded from this chart as it is the same for each level of Delay, and obfuscates the times for visualization and data transfer.

complete, the total cost was calculated by multiplying the total time by the total number of nodes listed in Table 6.

Figure 13 shows the total time per time step we observed for each of the isosurfacing plus rendering tests and Figure 14 shows the total time per time step

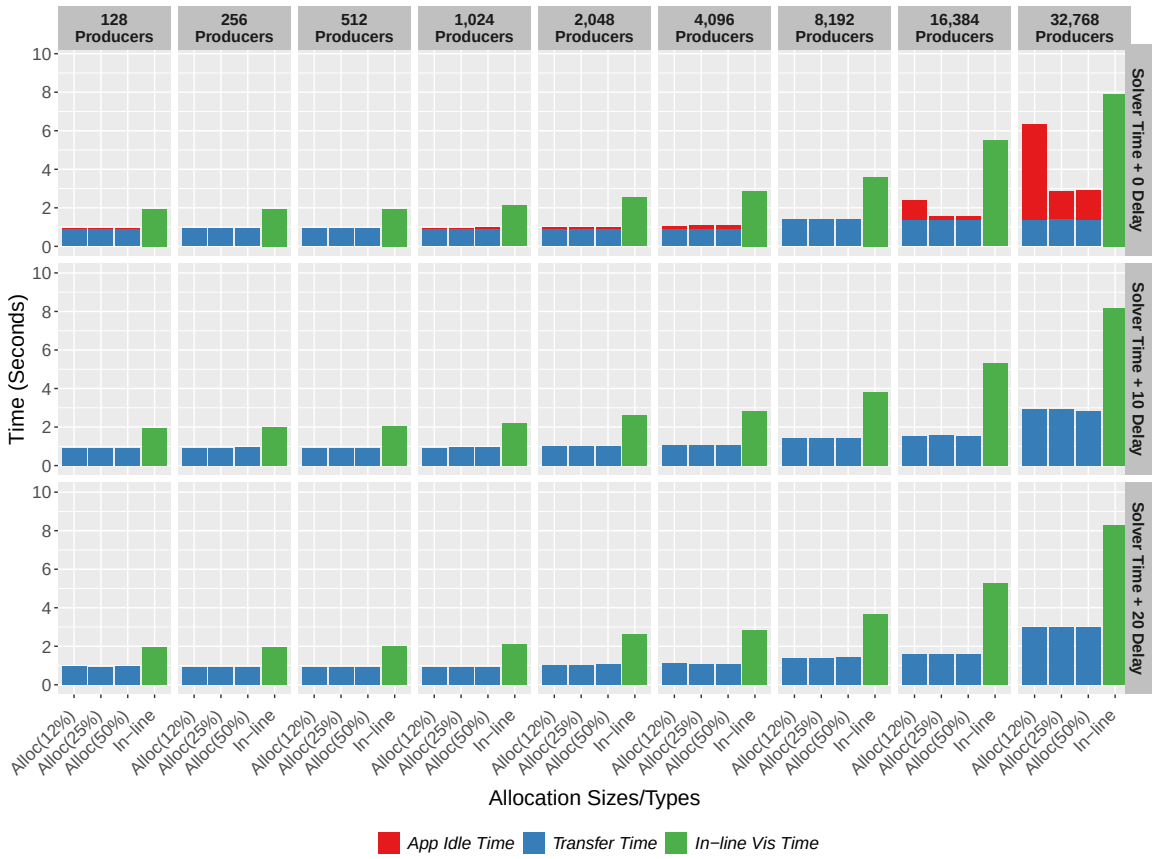


Figure 14. Stacked bar chart comparing the total time per step for using in-transit and in-line methods for volume rendering. This charts look at time from the applications perspective, meaning that the time for in-transit visualization is only how long it takes to transfer the data from the application, unless the in-transit resources block, in which case the application becomes idle. In-transit visualization is broken down into the time it takes to receive data data from the application and how long the application is blocked by the in-transit resources being too slow. In-line has a single time, how long it took to perform the visualization task. The application time is excluded from this chart as it is the same for each level of Delay, and obfuscates the times for visualization and data transfer.

we observed for each of the volume rendering tests. These charts break down the time of each step associated with each of our runs, showing if the application was blocked by the visualization and how long that blocking lasted, how long it took to transfer data from the application to the in-transit resources, and how long the in-line visualization operation took. With these charts it is easy to directly

compare the relative times of each of our different test configurations, and we can see very few cases where in-line took less time to use. This is especially apparent in the volume rendering tests, where in-transit was faster in every case. Conversely, we can see that there are cases for isosurfacing, where there was a fast simulation cycle time, where in-line was the fastest choice. This really highlights the need to understand the performance of visualization algorithms at all scales, as it would have been a better choice to skip visualizing some steps in those cases in order to not block the simulation.

Looking more in depth at the time charts, there are marked differences in the performance of the isosurfacing and rendering runs versus the volume rendering runs. The isosurfacing tests have large periods of blocking in the $Delay(0)$ cases, seen in the figure as *App Idle Cost*, whereas the volume rendering runs have very little. This observation further highlights the need to understand performance of visualization algorithms at different levels of concurrency, as the blocking time was cut by more than 50% in almost all cases when the in-transit resources were doubled from $Alloc(12\%)$ to $Alloc(25\%)$. Lastly, the in-line visualization times show an interesting trend as the application is scaled up. For isosurfacing, the in-line visualization times go up by nearly $7x$, while volume rendering only increases by approximately $4x$.

Figure 15 shows the total cost per time step we observed for each of the isosurfacing plus rendering tests and Figure 16 shows the total cost per time step we observed for each of the volume rendering tests. These charts break down the cost of each step associated with each of our runs, showing if the application was blocked by the visualization and how much that blocking cost, how much it cost to transfer data from the application to the in-transit resources, how long the

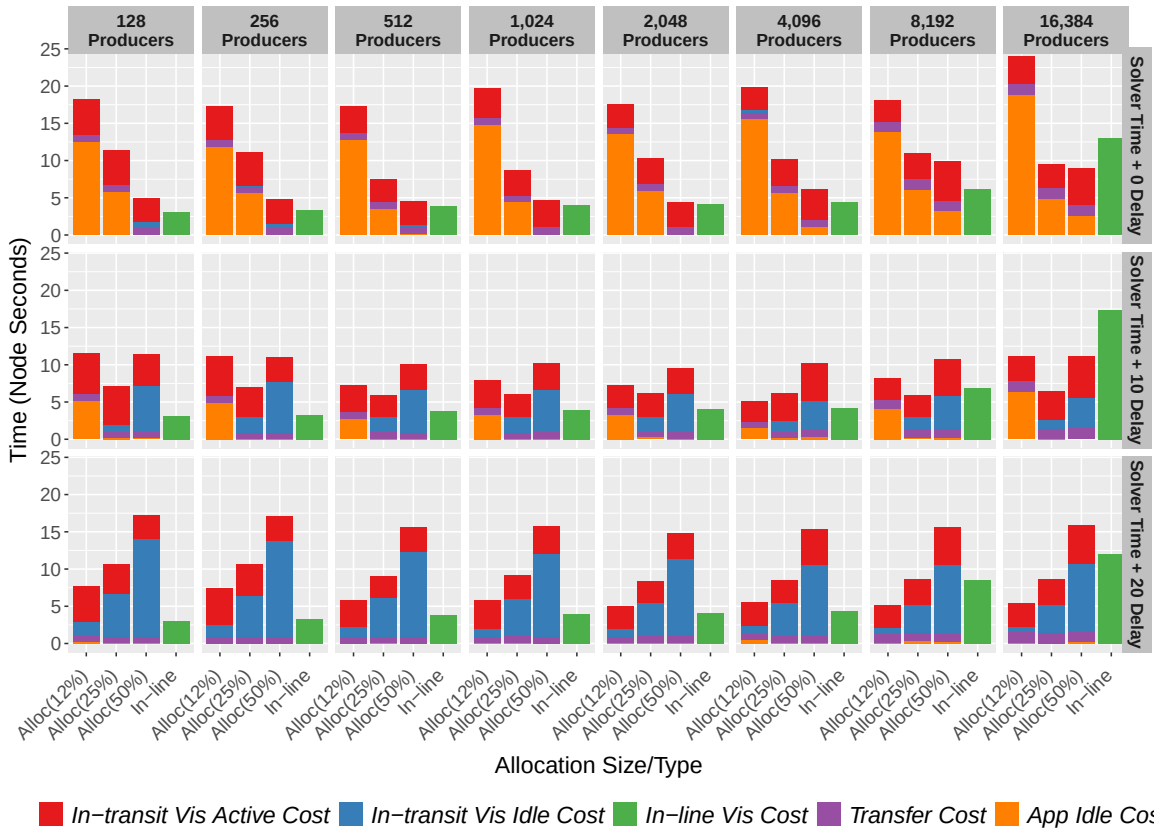


Figure 15. Stacked bar chart comparing the total cost per step for using in-transit and in-line methods for isosurfacing plus rendering. In-transit visualization is broken down into cost for the time that the visualization is actively working, cost for the time that it is idle, cost for the time it is receiving data from the application, and cost associated with blocking the application. The application active cost is excluded from this chart as it is the same for each level of Delay, and obfuscates the times for visualization and data transfer.

visualization resources were active and their cost, how long they were idle and that cost, and how long the in-line visualization operation took and its associated cost. With these charts it is easy to directly compare the relative costs of each of our different test configurations, and we can see many cases where in-line cost less to use. This is especially apparent in the isosurfacing plus rendering tests. Conversely, we can see that there are cases, especially at large-scale, where in-transit cost less, sometimes much less than the comparable in-line case. This really highlights the

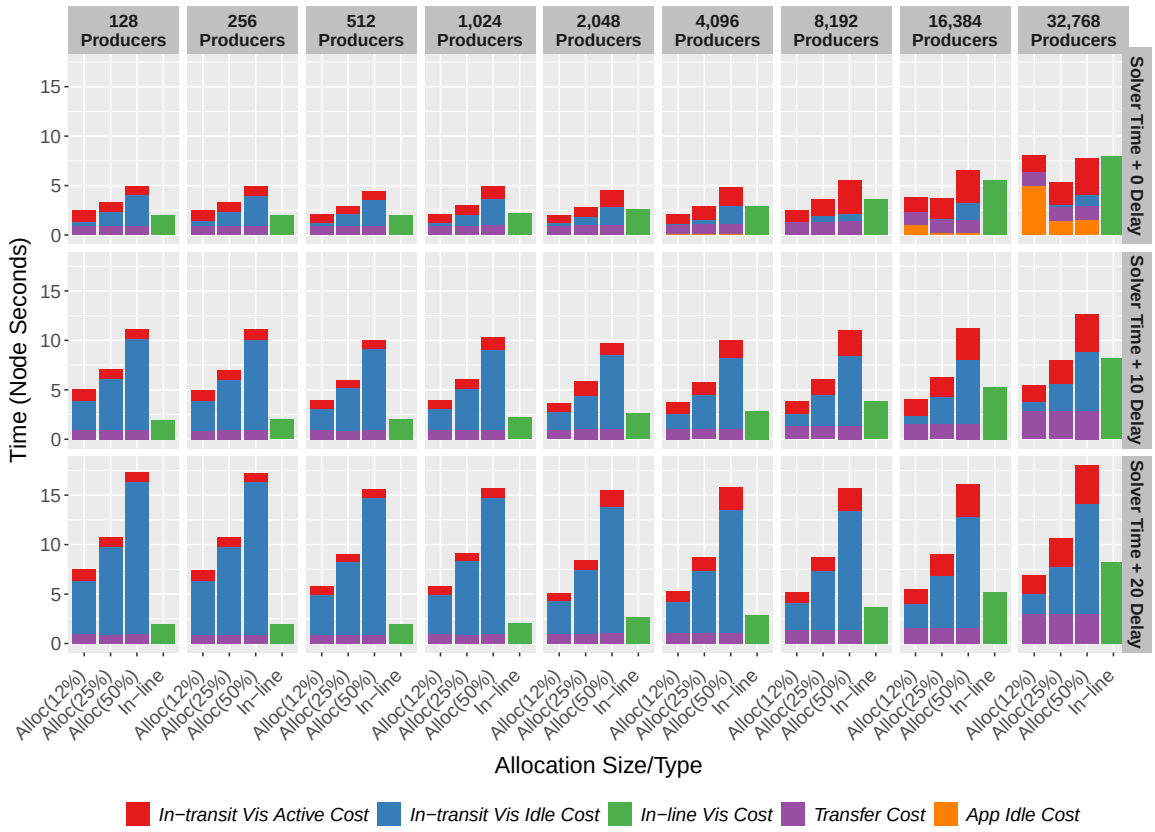


Figure 16. Stacked bar chart comparing the total cost per step for using in-transit and in-line methods for volume rendering. In-transit visualization is broken down into cost for the time that the visualization is actively working, cost for the time that it is idle, cost for the time it is receiving data from the application, and cost associated with blocking the application. The application active cost is excluded from this chart as it is the same for each level of Delay, and obfuscates the times for visualization and data transfer.

need to understand the performance of visualization algorithms at scale in order to choose the appropriate processing paradigm.

Looking more in depth at the cost charts, there are marked differences in the performance of the isosurfacing and rendering runs versus the volume rendering runs. The isosurfacing tests have large periods of blocking, seen in the figure as *App Idle Cost*, whereas the volume rendering runs have very little. One reason for that blocking was that on average, isosurfacing and rendering took twice as

long per step as volume rendering. Finally as the application cycle time increased, isosurfacing and rendering benefited more than volume rendering, showing that the isosurfacing tests were compute bound on the in-transit resources.

5.2 Summary

This chapter presented the data from a large scaling study ranging from 128 MPI tasks up to 32,768 MPI tasks. This scaling study used two different visualization algorithms, three different levels of in-transit allocation sizes, and a simulation with three distinct cycle times, for a total of 255 individual test runs. This data will be further analyzed in the following two chapters. Chapter VI will do a deep dive into the time to complete the visualization task for each of the configurations, and draw conclusions about configurations and scenarios which lend themselves to be more time efficient in-transit vs in-line. Chapter VII will do a deep dive into the costs for each of the configurations, and draw conclusions about configurations and scenarios which lend themselves to be more cost efficient in-transit vs in-line.

CHAPTER VI

TIME ANALYSIS

Most of the text in this chapter comes from [71], which was a collaboration between Scott Klasky (ORNL), David Pugmire (ORNL), Matthew Wolf (ORNL), Norbert Podhorszki (ORNL), Jong Choi (ORNL), Mark Kim (ORNL), Matthew Larsen (LLNL), Hank Childs (UO), and myself. The experimental infrastructure for this work was primarily constructed by myself. Matthew Larsen consulted on functionality and use of some of the infrastructure components. I ran all the experiments, compiled the results, created all of the initial analysis and figures, and wrote the majority of the manuscript text. David Pugmire was involved in helping to design the experiments, providing help and instruction for getting all of the tests completed, as well as editing of the final manuscript. Hank Childs provided extensive feedback during the work and was involved in editing the manuscript. Scott Klasky, Matthew Wolf, Norbert Podhorszki, Mark Kim, Jong Choi and Matthew Larsen were involved in initial discussions and provided comments on the final manuscript.

In this chapter we compare the time-to-solution for in-line and in-transit in situ visualization, analyzing when one paradigm is faster than another. To perform this comparison, we create a corpus of data by running a scaling study using two common visualization algorithms (isosurfacing and volume rendering), with in-line and in-transit. Our experiments vary an array of parameters, including the size of the in-transit in situ allocation, the simulation cycle time, and the scale of the simulation (up to 32,768 cores and 64 billion total cells). We then analyze this corpus of data and draw conclusions about when each paradigm is the fastest. Our findings show that in-transit is faster than in-line when inter-process

communication was high (up to 35% more efficient) and when our computation-bound algorithm was run at large scale (up to 47% more efficient). On the other hand, in-line was faster with our computation-bound algorithm and a short simulation cycle time (up to 42% more efficient). Finally, this work informs future directions for understanding other classes of in situ visualization algorithms.

6.1 Motivation

This chapter considers the goal of minimizing time-to-solution with in situ processing. There are multiple thrusts in HPC that motivate time-to-solution. One motivation for this problem includes "urgent HPC," i.e., real-time monitoring and fast turnaround. Examples include weather prediction [86], wildfires [92], hurricanes [68], earthquakes [58], and other catastrophic global events [31]. In these cases, fast in situ visualization helps the overall goal of each simulation. Another motivation is when domain scientists are actively studying the results (urgent HPC or otherwise) and would like to get visualizations as quickly as possible. One important use case within this latter motivation is the combination of simulation, observation, and experiment [55, 124]. Overall, these motivations form the fundamental premise behind our research: that in some cases domain scientists will want in situ visualization results as quickly as possible.

In this chapter we present a study comparing time-to-solution for in-line and in-transit in situ visualization, measuring impact on the ability of the simulation to progress quickly. Our contributions from this study inform desirable in situ configurations across a variety of simulation scales for both a computation-bound and communication-bound visualization operation.

6.2 Related Works

Several studies already exist that consider in-transit and in-line from the perspective of time-to-solution. Morozov et al. [105] describes a system for launching in situ/in-transit analysis routines, and compares each in situ technique based on time to solution for two different analysis operations. Friesen et al. [53] describes a setup where in-line and in-transit visualization are used in conjunction with a cosmological code to run two different analysis routines. Bennett et al. [22] use both in-line and in-transit techniques for analysis and visualization of a turbulent combustion code. Ayachit et al. [18] performed a study of the overheads associate with using the generic SENSEI data interface to perform in situ analysis using both in-line and in-transit methods. The common theme between these and other studies is that they primarily consider analysis pipelines, which can have different communication and computation overheads versus visualization pipelines.

Our work approaches the problem differently than these past works. First, we concentrate on in situ visualization pipelines. Second, we focus specifically on in-line in situ vs. in-transit in situ from the perspective of simulation cycle time, visualization type, resource requirements, and how different combinations of these factors impact the final time-to-solution of the simulation.

There are three highly relevant works preceding this work:

- Oldfield et al. [111] consider in-transit and in-line times for analysis tasks, but only see a small margin of cases where in-transit is faster, due to the scaling characteristics of the algorithms they studied. As such our findings are complimentary to theirs in terms of the algorithms studied.
- Malakar et al. did twin studies on cost models, one for in-line [90] and one for in-transit [89]. Once again, these studies did not consider optimizing the

time-to-solution. Further, they considered optimizing analysis frequencies and resource allocations, which is complementary to our effort.

– The authors of this paper considered tradeoffs between in transit and inline in a previous work [72]. Our previous study showed strong evidence for in-transit time savings for the simulation. However, the algorithm considered was computation-heavy, so the extent of the effect was smaller. The current paper focuses exclusively on time savings, considering both computation- and communication-heavy visualization algorithms. Finally, we note our corpus of data for this study in part draws on runs from that study.

6.3 Factors Affecting Time-to-Solution

The primary drawback of incorporating in situ visualization routines into a simulation code are the negative effects on the simulation's runtime. In-line visualization pauses the simulation while the visualization completes. For fast visualization operations this impact may be minimal, but it also may be prohibitive for slower communication-heavy operations. Conversely, in-transit pauses the simulation while the data is being transferred from the simulation nodes to the visualization nodes. This pause can be short or long, depending upon a number of factors. The pause will be shorter if the visualization nodes are ready to receive data as soon as the simulation completes a step and is ready to transmit data. The pause will be longer if the simulation completes a step and the visualization nodes are still busy finishing operations on the previous time step. In this case, the simulation will have to wait for the visualization nodes to finish, and then transfer the data, incurring a larger time penalty. Pausing the simulation here is not the only possible scenario, but it is the choice we made in the context of this study, as we did not want to loose simulation time steps.

Given that each in situ paradigm necessitates pausing the simulation to some degree, the paradigm that blocks the least will have a faster run time. In order for in-line to have the smallest impact on overall runtime, the visualization needs to scale well at the concurrency level of the simulation. In order for in-transit to have the smallest impact on overall runtime, the data transfer needs to be fast, and the visualization needs to scale well at the concurrency level of the smaller in-transit allocation. In this work, we focus our scope to the execution time of visualization operations both in-transit and in-line. Our goal is to evaluate the performance of different visualization algorithms across a variety of simulation and in-transit allocation sizes.

Our hypotheses in this chapter is that there should be clear distinctions between scenarios in which an algorithm performs well in-line or in-transit, and a major contribution of this work is confirmation of that hypothesis In order to confirm our hypothesis, we ran 255 individual in situ workflows, covering a range of concurrencies (up to 32,768 cores), simulation cycle times (5, 15, and 25 seconds), in situ configuration (in-line, and in-transit), and two visualization workflows, one that was computation-bound (isosurfacing) and one that was communication-bound (volume rendering). From these experiments we found that our communication-bound workload ran faster in-transit versus in-line, in all cases. In addition, we found that our computation-bound workload was faster in-line in many cases with a short simulation cycle time; as simulation cycle time increased however, in-transit became faster. Further contributions of this work include a detailed analysis of when to choose in-line or in-transit in situ by comparing algorithm performance across a variety of simulation cycle times and in-line and in-transit allocation sizes.

6.3.1 Conceptual Timing Scenarios for In Situ Visualization.

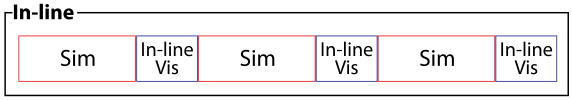
When integrating in situ visualization, a key question is which type to use (in-line or in-transit). Here, we will discuss two primary things: (1) factors that influence one paradigm being more time efficient than the other, (2) the effect of those factors on in-line versus in-transit workflows. Figure 17 contains the four example scenarios we will discuss. Each of these scenarios will demonstrate ways in which both in-line and in-transit in situ can succeed or fail at being time efficient.

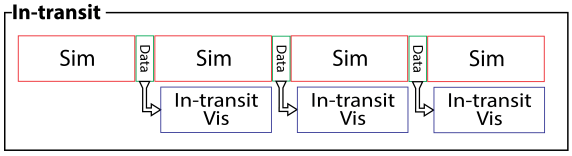
6.3.1.1 Scenario 1: In-transit Data Transfer is Fast.

Figure 17a depicts this scenario. In this case the time it takes to transfer data to the in-transit resources is less than the corresponding time for in-line visualization. In addition, the in-transit visualization time is larger than that of in-line, but it does not impact the simulation because it is performed asynchronously and the resources are ready for the next step by the time the simulation is ready to perform the next write. In this example, moving the data off of the simulation nodes results in an overall faster time to solution, allowing 4 steps to be completed in-transit when in-line only completed three.

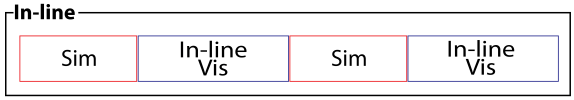
6.3.1.2 Scenario 2: In-transit Visualization Scales Better.

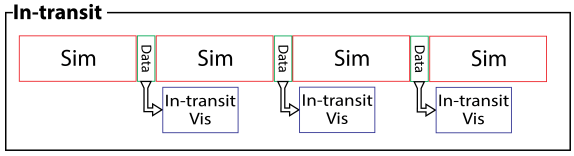
Figure 17b depicts this scenario. In this case the time it takes to perform in-transit visualization is roughly half of that of in-line visualization. This speaks to the scalability of the visualization algorithm itself. The time to solution was larger in-line because the algorithm was communication bound, whereas on the in-transit resources the communication bottleneck was sidestepped by operating on fewer nodes, completing much faster. In this example, faster visualization allowed for twice as many simulation cycles to be completed vs. that of in-line.



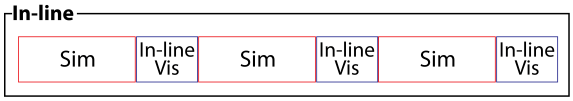


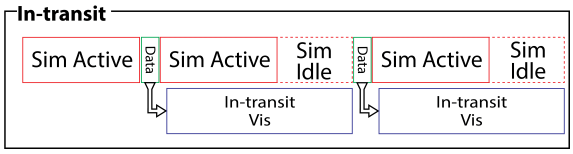
(a) Here, the data transfer for in-transit is faster than the visualization step for in-line, meaning the in-transit simulation can advance more quickly.



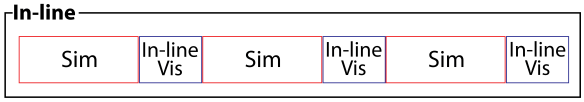


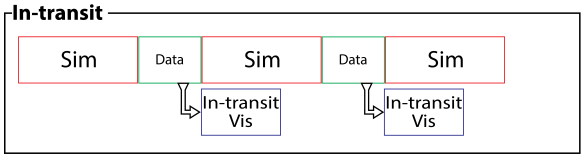
(b) Here, the in-transit visualization exhibits better scaling than the in-line visualization, meaning in-transit is more efficient on fewer resources.





(c) Here, the in-line visualization exhibits better scaling that the in-transit visualization, meaning that in-transit was compute bound.





(d) Here, data transfer for in-transit takes as long as the corresponding in-line visualization cycle, meaning in-transit can never be more time efficient.

Figure 17. Gantt charts showing possible scenarios of how the simulation and visualization could progress over time (from left to right) with both in-line and in-transit in situ. Each example will show a possible scenario and the effects to the simulation and overall time.

6.3.1.3 Scenario 3: In-line Visualization Scales Better.

Figure 17c depicts this scenario. In this case the time to perform visualization in-transit takes three times as long as in-line. This speaks to the scalability of the visualization algorithm itself. The time-to-solution was larger in-transit because the algorithm was compute bound, whereas on the in-line resources the compute bottleneck was sidestepped by operating at the full scale of the simulation, completing much faster. In this example, faster in-line visualization allowed for

three visualization cycles to be completed in-line, with only two completed in-transit.

6.3.1.4 Scenario 4: In-transit Data Transfer is Slow.

Figure 17d depicts this scenario. In this case the time it takes to transfer data to the in-transit resources is equal to the time it takes to perform the in-line visualization task. Even though the in-transit visualization time is small enough to not block the simulation once it reaches the next step, the long transfer time has eliminated all chance of saving time in-transit. This example speaks to the challenge that in-transit visualization faces, it can only achieve a time savings if the data transfer time is fast, giving it a relatively small window to be efficient. In this case, due to the transfer time and visualization time being equal, both in-transit and in-line advance the simulation to the same place.

6.4 Results

The objective of our experiments was to understand the performance of both a computation-bound, and a communication-bound, visualization algorithm both in-transit and in-line. Our results focus on in situ time-to-solution. First, we evaluate factors that limit in situ speed-ups in Section 6.4.1. Second, we determine whether in-transit in situ can keep up with the pace of the simulation, focusing on in-transit allocation size, workload, and simulation frequency in Section 6.4.2.

6.4.1 Breaking Down the Time-to-Solution. In this section we will look at the limiting factors for in situ visualization in the context of both in-transit and in-line paradigms. We will first analyze the breakdowns of where each in situ method spent its time in each of our experiments, and draw conclusions about performance under varying visualization workload and system constraints. We will then explore why in-line rendering performs poorly in comparison to in-

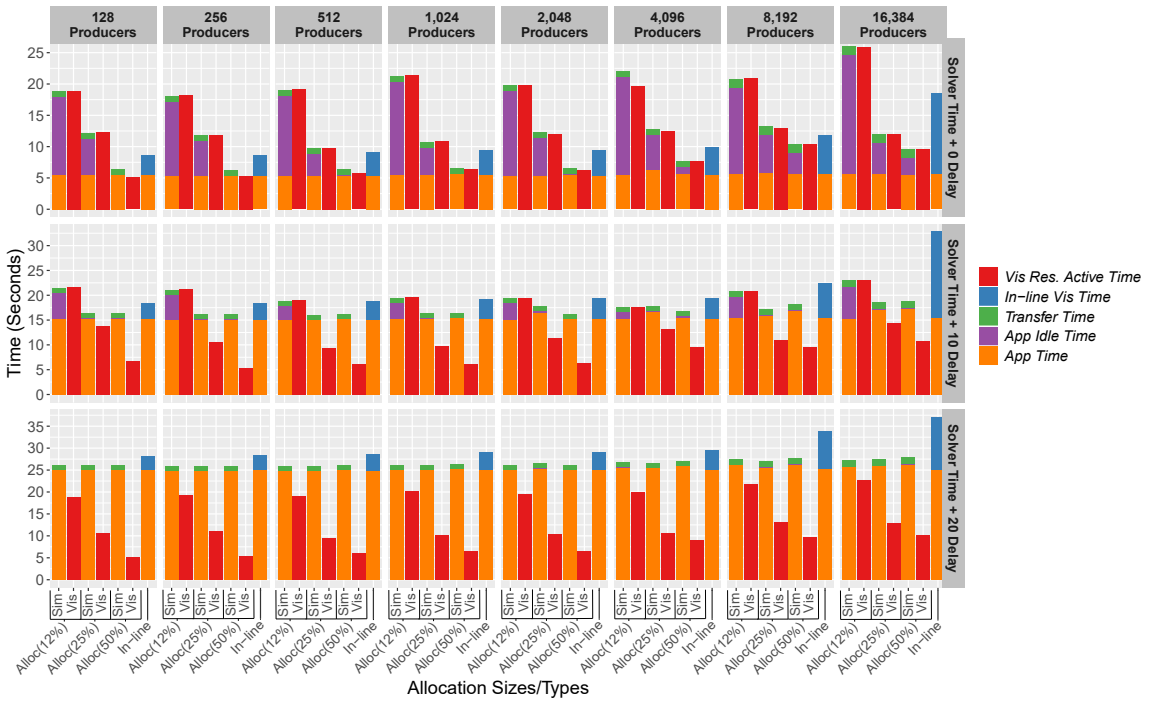


Figure 18. Comparing the total time per step for using in-transit and in-line methods for isosurfacing plus rendering. This chart looks at time from the applications perspective, meaning that the time for in-transit visualization is only how long it takes to transfer the data from the application, unless the in-transit resources block, in which case the application becomes idle. In-transit visualization is broken down into the time it takes to receive data data from the application and how long the application is blocked by the in-transit resources being too slow. A second column is present for each in-transit case that shows how long the in-transit resources were active during a single time step, giving a better sense of where blocking and idle time occurs. In-line has a single time, how long it took to perform visualization.

transit rendering, and how in-transit in situ can effectively take advantage of this performance characteristic.

Figure 18 shows the total time per time step we observed for each of the isosurfacing plus rendering experiments and Figure 19 shows the total time per time step we observed for each of the volume rendering tests. The times on these charts are broken down into the following categories: (1) **Orange Bars:** the time that the application active. (2) **Purple Bars:** the time that the application was

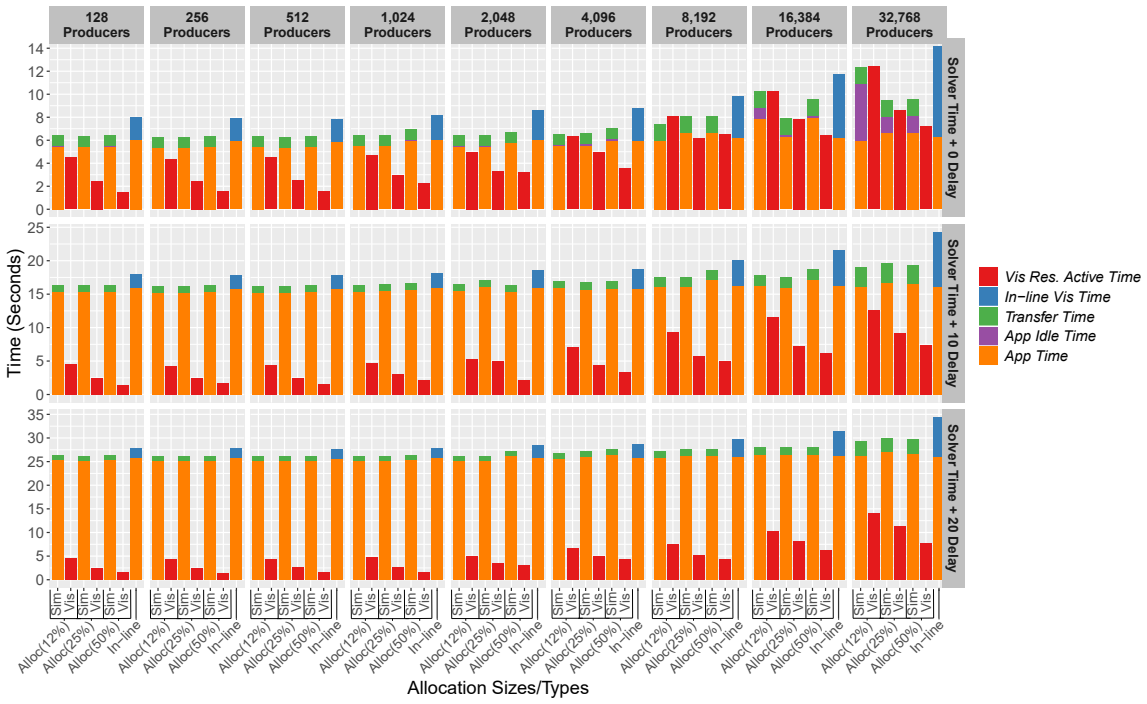


Figure 19. Comparing the total time per step for using in-transit and in-line methods for volume rendering. This chart looks at time from the applications perspective, meaning that the time for in-transit visualization is only how long it takes to transfer the data from the application, unless the in-transit resources block, in which case the application becomes idle. In-transit visualization is broken down into the time it takes to receive data data from the application and how long the application is blocked by the in-transit resources being too slow. A second column is present for each in-transit case that shows how long the in-transit resources were active during a single time step, giving a better sense of where blocking and idle time occurs. In-line has a single time, how long it took to perform visualization.

idle. (3) **Green Bars:** the time to transfer data to the in-transit resources. (4) **Red Bars:** the time to perform in-transit visualization. (5) **Blue Bars:** the time to perform in-line visualization.

Now that the basic elements of the charts have been explained, we will discuss the timings for our isosurfacing experiments from Figure 18. The first thing to notice in this chart, is the poor performance of in-transit in situ in the *Alloc*(12%) and *Alloc*(25%) experiments at *Delay*(0). All of these experiments

have large portions of time where the simulation is blocked because the in-transit visualization was unable to keep up with the simulation. This blocking effect made it so that in-line in situ was the most performant choice for all but the largest scale (16,384 processes). Another observation from this chart is the absence of simulation blocking in all of the $Delay(20)$ cases. In each of those cases, the only time delay for the simulation was the time to transfer data to the in-transit resources. However, these case also showcase the other negative of in-transit visualization, idle in-transit resources. In every case, the in-transit resources were idle for some percentage of the simulation cycle, the worst being $Alloc(50\%)$, which was idle for up to 80% of each simulation cycle. This level of idle time means that resources were severely over allocated, and either the resources need to be reduced, or the visualization pipeline needs to be adaptive, and to dynamically add new visualization operations to the queue in order to make productive use of the allocated resources.

Next, we will discuss the timings for our volume rendering experiments in Figure 19. The performance of in-transit volume rendering is significantly different from isosurfacing. In most cases in-transit volume rendering was faster than the simulation cycle time. Only at the two larges cases (16,384 and 32,768 procs) with the $Delay(0)$ caused the simulation to block. This difference is because volume rendering was communication bound, and is more efficient at smaller scale. This fact led in-transit volume rendering to be faster than in-line in every single experiment we performed. This success at not blocking the simulation also had a pitfall however, which was idle in-transit resources. As evidenced by the large idle times (up to 88% of the total runtime), some of the allocations were too large. We

could have achieved similar time-to-solution by using fewer resources, which would have reduced the resource idle time.

Finally, taking what we learned from both Figure 18 and Figure 19, we will compare and contrast differences to provide general guidelines for which in situ method will perform the best with a given workload and concurrency. The first point to raise is that there are very few cases where in-line took less time to use. This is especially apparent in the volume rendering tests, where in-transit was faster in every case. Conversely, we can see that there are cases for isosurfacing, where there was a fast simulation cycle time, where in-line was the fastest choice.

Looking more in depth at the time charts, there are marked differences between the performance of the isosurfacing and volume rendering runs. The isosurfacing tests have large periods of blocking in the $Delay(0)$ cases, seen in the figure as *App Idle Cost*, whereas the volume rendering runs have very little. This observation further highlights the need to understand performance of visualization algorithms at different levels of concurrency, as the blocking time was cut by more than 50% in almost all cases when the in-transit resources were doubled from $Alloc(12\%)$ to $Alloc(25\%)$. Lastly, the in-line visualization times show an interesting trend as the application is scaled up. For isosurfacing, the in-line visualization times go up by nearly *7x*, while volume rendering only increases by approximately *4x*.

In order to better understand these visualization performance differences, we look specifically at the cumulative rendering and compositing times in Figure 20. Rendering scales very well up through $4,096$ processes. Beyond that, the communication at higher levels of concurrency leads to a drop in scalability for in-line isosurfacing and volume rendering. Isosurfacing for example, has the

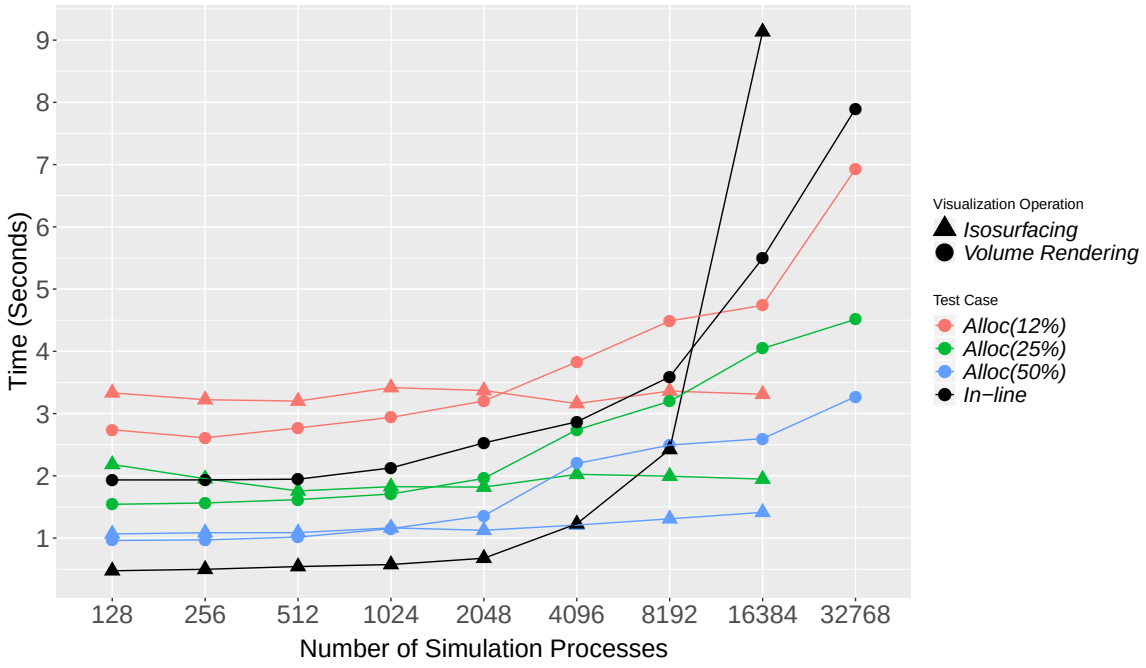


Figure 20. Total time per step to render and composite an image, both in-transit and in-line. The results from isosurfacing (triangles) and volume rendering (circles) are shown. Experiments are grouped by color (configuration) and connected by lines (concurrency sequence).

compositing and rendering time rise from 1 second per step at $4,096$ processes up to 9 seconds at $16,384$ processes; an increase of *9x*. Volume rendering has a much smaller rise in time, from 2.5 seconds at $4,096$ processes, up to 5.5 seconds at $16,384$ processes; an increase of about *2.5x*. In summary, these trends show how in-line scalability problems create an opportunity for in-transit to be faster, especially at scale.

6.4.2 Can In-transit Visualization Keep Up?. In this section we will look at whether in-transit in situ visualization if fast enough to not block the simulation. First, we will explore the differences in the time to move the data from the simulation to the in-transit resources compared against the time to perform in-line visualization. Second, we will analyze the cases where in-transit was able

to keep up. Finally, we will compare how home many simulation steps both in-line and in-transit can complete in 500 seconds.

Figure 21 highlights the differences in time to perform an in-transit data transfer vs. the time it takes to perform in-line visualization for both isosurfacing and volume rendering. In this figure we see that there are no instances where the data transfer takes more time than the associated in-line visualization operation. This means that for the algorithms we tested, in-transit visualization always has a chance to be faster than in-line visualization. The other main feature to see in this figure is the widening gap between data transfer time and the comparable in-line visualization times as scale increases. This widening gap helps to explain how in-transit isosurfacing was able to become faster than in-line at $16,384$ processes (see Figure 18).

Figure 22 highlights a important feature of in-transit visualization, that it is difficult to both keep the in-transit resources busy for an entire simulation cycle while also not blocking the simulation. This figure shows how long the in-transit resources were idle each simulation cycle in relation to the length of a simulation cycle. For example, in order for the in-transit idle percentage to be 0%, the simulation and visualization cycle time would need to be the same. If the in-transit idle percentage is -100% idle, the visualization would be $2X$ the simulation cycle time.

Looking at the $Delay(0)$ column of Figure 22, we can see that there is a large variation in the idle times for in-transit. In the worst case (from the simulation's perspective) in-transit blocks the simulation from proceeding for 3.5 (-350%) simulation cycles. While the best cases have in-transit being neither idle, nor blocking the simulation This trend highlights the effects of simulation

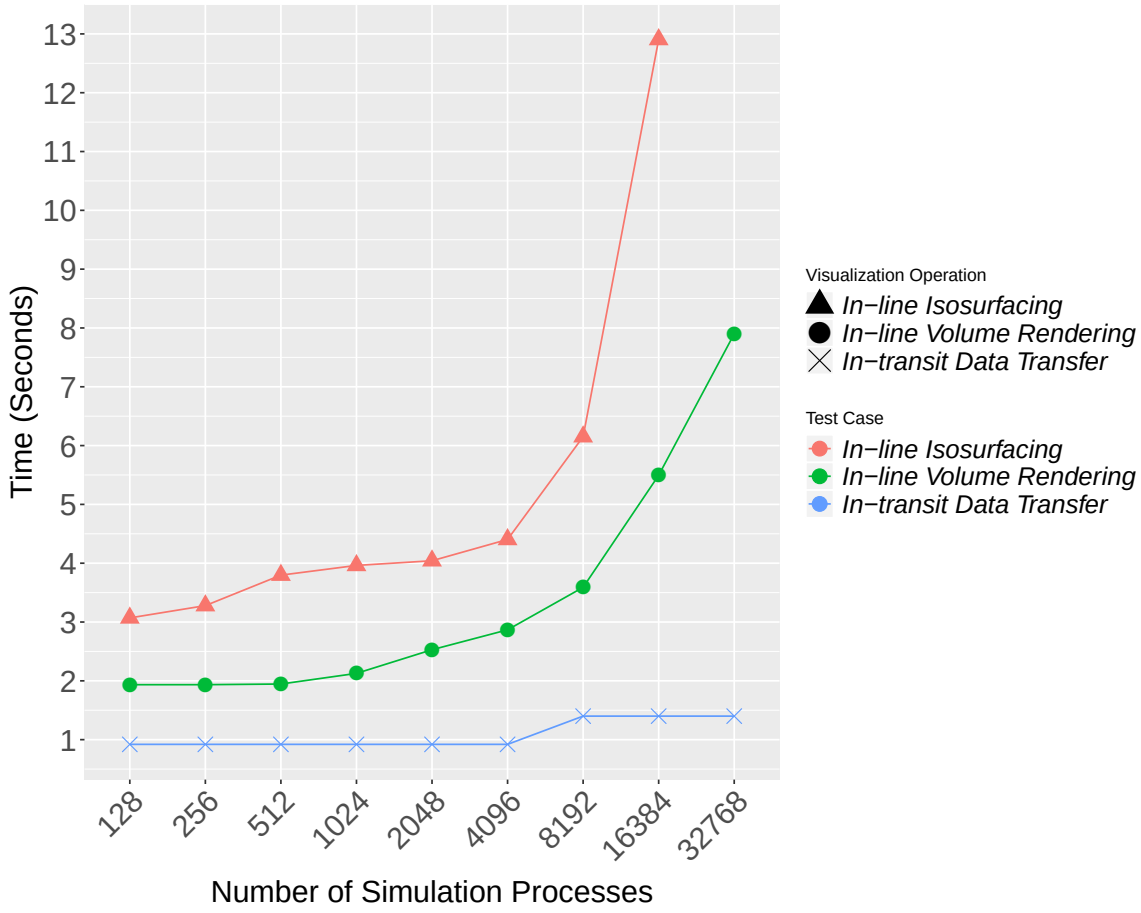


Figure 21. Total time per step for in-transit in situ to transfer data off of the simulation nodes compared against the time that in-line takes to perform either the isosurfacing or volume rendering operations. In essence, this chart shows how long the simulation had to pause each simulation step for visualization to take place, either in-line, or in-transit by moving the data to a separate allocation and performing the visualization asynchronously to the simulation.

cycle time on the size of the in-transit resources necessary to complete a task in time. Looking at the $Delay(10)$ and $Delay(20)$ columns, almost every case was able to complete without blocking the simulation. Overall, the volume rendering runs were less adversely affected by simulation cycle time, further highlighting the performance differences between compute and communication bound visualization algorithms.

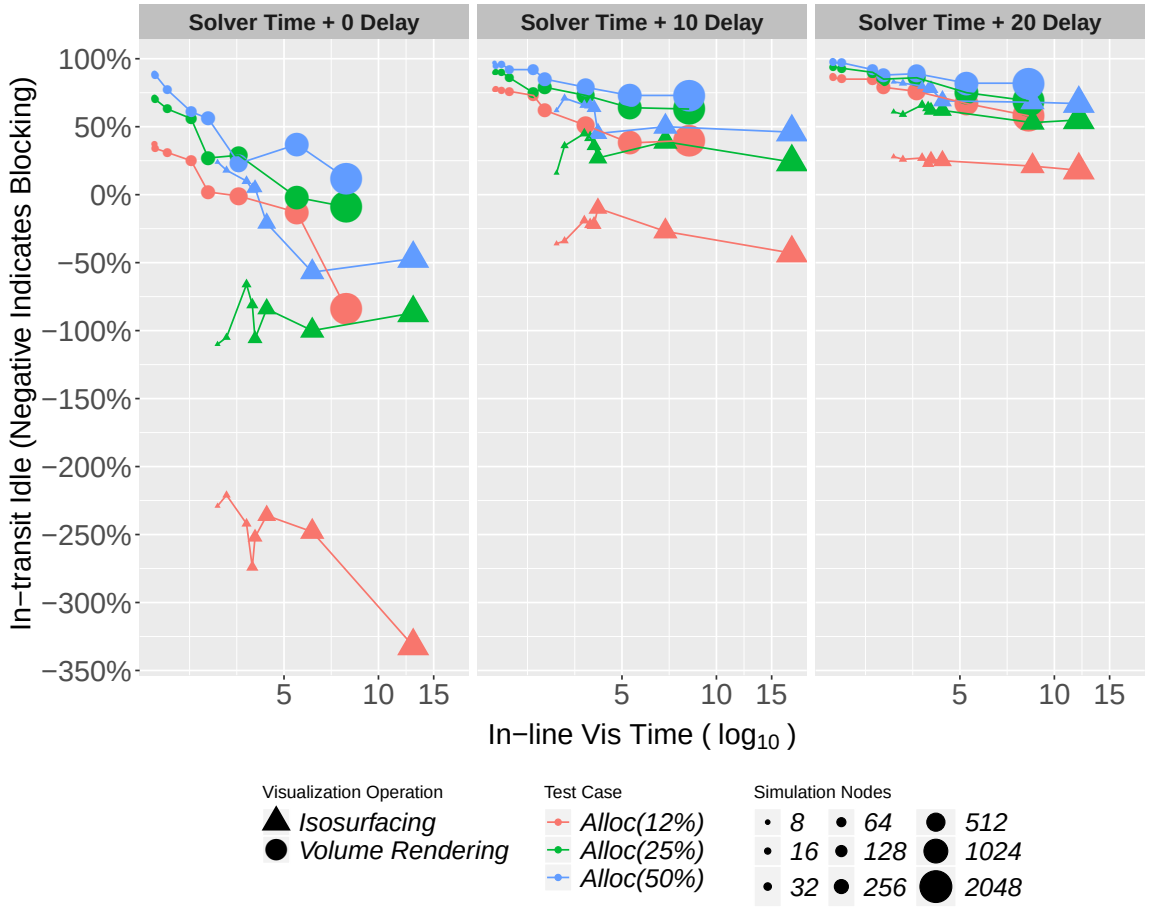


Figure 22. An analysis of whether in-transit resources idle or blocking during the course of the simulation. The y-axis shows how idle the in-transit resources were during each simulation cycle. An idle time of 0% indicates in-transit resources were always busy and never blocked. 100% idle means the in-transit resources were always idle. While -100% idle means that the in-transit resources blocked the simulation from proceeding for an entire simulation cycle. The results from in-transit isosurfacing (triangles) and volume rendering (circles) are shown. Each glyph is scaled by the concurrency of the experiment (isosurfacing: 8-1024; volume rendering: 8-2048). Experiments are grouped by color (configuration) and connected by lines (concurrency sequence).

Figure 23 is a shows the impact of visualization on the progress of the simulation. Given a fixed time allocation of 500 seconds, the graphs show how many simulation time steps can be completed with and without visualization. The case where no visualization is performed is the high water mark for each graph.

For example, a $Delay(0)$ volume rendering configuration with $16,384$ simulation processors can complete 43 cycles using in-line visualization and 64 cycles using in-transit ($Alloc(25\%)$). This means that a 25% increase in resources led to a 48% increase in productivity ($64/43 \times 100\% - 100\%$). Similarly, for isosurfacing, $Delay(0)$ and $Alloc(50\%)$ yields a 100% increase in productivity (26 cycles to 52 cycles) for 50% more resources, $Delay(10)$ and $Alloc(12\%)$ yields a 46% increase (15 cycles to 22 cycles) for 12% more resources, and $Delay(10)$ and $Alloc(25\%)$ yields an 80% increase (15 cycles to 27 cycles) for 25% more resources. In summary, this chart shows that depending upon the length of the simulation cycle, a large increase in productivity is possible (50% or more in some configurations), with only a small increase in resource allocation (as small as 12% more).

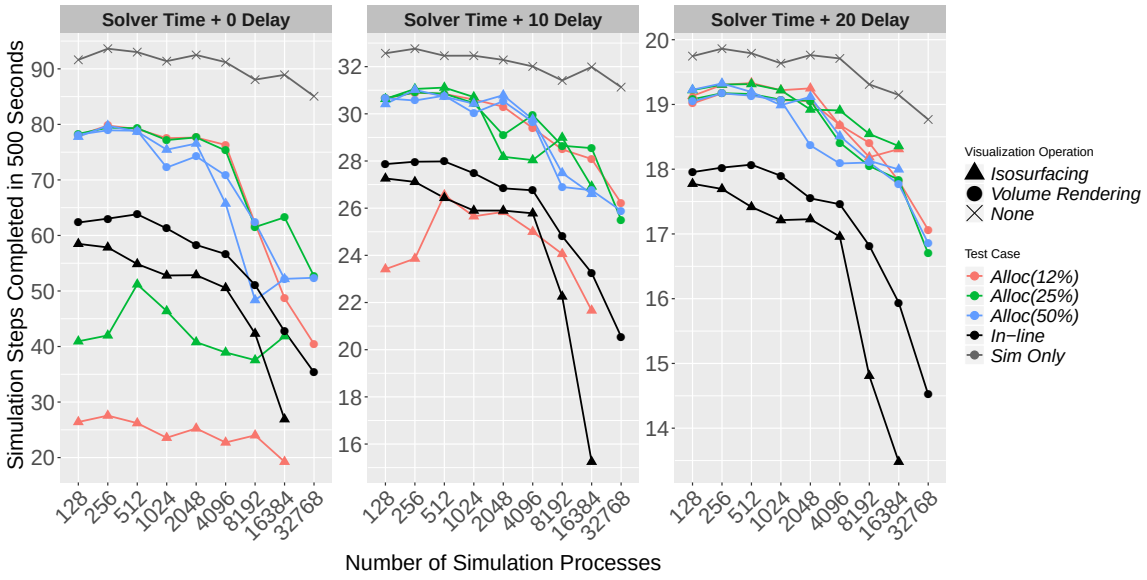


Figure 23. The number of simulation steps that can be completed in a 500 second time budget for each in situ configuration. Higher numbers are better, meaning that more simulation cycles could be completed within the time limit. The results from isosurfacing (triangles) and volume rendering (circles) are shown for both in-transit and in-line, along with a reference line were no visualization was done. Experiments are grouped by color (configuration) and connected by lines (concurrency sequence).

6.5 Summary

In this chapter we have presented a study that compares the time-to-solution for in-line and in-transit in situ visualization, and provide an analysis of when and why one paradigm is faster than another. We believe that for use cases where time-is-of-the-essence, these insights will help simulation teams run on increasingly larger computing resources. Without these insights, simulation teams run the risk of performing an overly limited set of visualizations out of an abundance of caution, or choosing the an in situ paradigm that results in slower times. With this work we have analyzed the time-to-solution performance of two common visualization algorithms at scale. Our experiments gave insight into our hypothesis presented in Section 6.3, and we were able to show that the two classes of algorithms we studied did indeed exhibit different scaling properties. We found that for our communication-bound algorithm (volume rendering), in-transit in situ enabled the simulation to advance further in a majority of cases vs in-line, especially at large scale. We also found that our computation-bound algorithm (isosurfacing), in-line had better performance, especially at smaller scale.

This paper highlights several interesting areas of future work. First, this paper explored two different types of algorithms. These two algorithms exhibited different behaviors which helped us to understand these two classes of algorithms. We intend to study additional classes of algorithms to understand the behavior and performance at scale using different in situ paradigms. Second, studying the implications of even larger scale is required. Timings started to change significantly for in-line at the largest scales. Examination of these trends at higher scales will provide additional insight into in situ visualization. Finally, the *Alloc* sizes chosen in this study were much too large for some of our experiments. Studying lower

percentages of in-transit allocations will help to reduce resource requirements for future in situ integrations, while also ensuring a fast time-to-solution.

CHAPTER VII

COST ANALYSIS

Most of the text in this chapter comes from [73], which was a collaboration between Scott Klasky (ORNL), David Pugmire (ORNL), Matthew Wolf (ORNL), Norbert Podhorszki (ORNL), Jong Choi (ORNL), Mark Kim (ORNL), Matthew Larsen (LLNL), Hank Childs (UO), and myself. The experimental infrastructure for this work was primarily constructed by myself. Matthew Larsen consulted on functionality and use of some of the infrastructure components. I ran all the experiments, compiled the results, created all of the initial analysis and figures, and did a significant amount of writing for the manuscript. David Pugmire was involved in helping to design the experiments, providing help and instruction for getting all of the tests completed, as well as being heavily involved in editing of the final manuscript. Hank Childs provided extensive feedback during the work and had a major hand in writing sections of the report. Scott Klasky, Matthew Wolf, Norbert Podhorszki, Mark Kim, Jong Choi and Matthew Larsen were involved in initial discussions and provided comments on the final manuscript.

In this chapter we analyze the opportunities for in-transit visualization to provide cost savings compared to in-line visualization. We begin by developing a cost model that includes factors related to both in-line and in-transit which allows comparisons to be made between the two methods. We then run a series of studies to create a corpus of data for our model. We run two different visualization algorithms, one that is computation heavy and one that is communication heavy with concurrencies up to $32,768$ cores. Our primary results are in exploring the cost model within the context of our corpus. Our findings show that in-transit consistently achieves significant cost efficiencies by running visualization algorithms

at lower concurrency, and that in many cases these efficiencies are enough to offset other costs (transfer, blocking, and additional nodes) to be cost effective overall. Finally, this work informs future studies, which can focus on choosing ideal configurations for in-transit processing that can consistently achieve cost efficiencies. We start by motivating the benefits and findings of our cost model, discuss related works, describe the details of the cost model, and then discuss our findings and their implications.

7.1 Motivation

In-transit visualization incurs new costs that do not exist with in-line visualization. There are additional resources for the in-transit nodes, and a new activity to perform: transferring the data from the simulation nodes to the in-transit nodes. Further, if the in-transit nodes are not able to perform their tasks quickly enough, they can block the simulation from advancing. (Blocking the simulation is not the only possible decision for this scenario, but it is the decision we consider in the context of this paper.)

Despite these additional costs, in-transit also has a potential cost advantage that in-line does not have. The number of in-transit nodes is typically much less than the simulation nodes. Further, when algorithms exhibit poor scaling, fewer nodes are more efficient. In effect, in-transit has the potential to reduce costs that result from poor scaling of visualization algorithms. Consider a scenario: if a visualization algorithm takes 1 second on 1000 nodes running in-line, but the same algorithm takes 50 seconds on 10 nodes running in-transit, then the visualization cost is 1000 node seconds for in-line and 500 node seconds for in-transit. We define a term to capture this phenomenon: **Visualization Cost Efficiency Factor (VCEF)**. VCEF is the in-line visualization cost divided by the in-transit

visualization cost. In the scenario just described, the VCEF would be 1000/500 or 2 — the cost to perform in-line is 2X more than in-transit. Of course, VCEF is just one consideration, alongside the other (unhelpful for cost savings) factors for in-transit: extra resources, transfer costs, and blocking.

Our hypothesis entering this study is that there are configurations of in-transit visualization such that the cost to reach the final solution are less in-transit than in-line. To that end, for this study, we consider the topic of relative costs between in-transit and in-line visualization. What makes our study novel is the identification and usage of VCEF. We observe that VCEF is a significant phenomenon; our communication-heavy algorithm regularly yields a VCEF of four or above, and even our computation-heavy algorithm yields such values at very high concurrency. This high VCEF value in turn allows in-transit to become cost effective overall in many scenarios, as the savings are enough to offset other costs. We also provide a model for reasoning about this space, and a corpus of data that reflects experiment times for currently popular software. Overall, this study provides significant evidence that in-transit can be cost effective.

7.2 Related Works

There are three highly relevant works preceding this work:

- Oldfield et al. [111] also considered in-transit and in-line costs. The main difference between their work and our own is that they focused on analysis tasks which did not benefit from a $VCEF$ speedup. As such, their findings differ from ours.
- Malakar et al. did twin studies on cost models, one for in-line [90] and one for in-transit [89]. Once again, these studies did not consider $VCEF$. Further,

they considered optimizing allocation sizes and analysis frequencies which is a complementary task to our effort.

– Work by Kress et al. [72] considered trade-offs between in-transit and in-line for isosurfacing at high concurrencies. This study was the first to show evidence of $VCEF$. However, the algorithm considered was computation-heavy, so the extent of the effect was smaller and only appeared at very high concurrency. Further, that paper lacked a cost modeling component, rather just observing that the phenomenon was possible. Our paper focuses exclusively on cost savings, providing a model and considering both computation- and communication-heavy visualization algorithms. Finally, we note the corpus of data for our study in part draws on runs from the Kress et al. study.

7.3 Cost Model

This section defines a cost model for determining when in-transit visualization can cost less than in-line visualization. First, terms are introduced for the operations that occur in both in-line and in-transit visualization. Next, we use those terms to demonstrate when in-transit will cost less than in-line visualization, and provide a discussion for when and how this occurs. Finally, we derive a formulation to determine the degree of scalability of in-transit over in-line, (VCEF), that is required for in-transit to be cost effective.

7.3.1 Definition of Terms. Below we define terms for both in-line and in-transit visualization operations.

– Let T be the time for the simulation to advance one cycle.

– Let N be the number of nodes used by the simulation code.

- Let Res_p be the proportion of nodes (resources) used for in-transit visualization. E.g., if the number of nodes for the simulation (N) is 10,000 and the number of nodes for in-transit visualization is 1,000, then $Res_p = 1,000/10,000$, which is 0.1.

- Let Vis_p be the proportion of time spent doing visualization in the in-line visualization case. E.g., if T is 5 seconds and the in-line visualization time is 1 second, then $Vis_p = 1/5$, which is 0.2.

- Let $Block_p$ be the proportion of time that the simulation code is blocking while waiting for in-transit visualization to complete. E.g., if T is 5 seconds and the simulation has to wait an additional 2 seconds for the in-transit resources to complete, then $Block_p = 2/5$, which is 0.4. If the in-transit visualization completes and does not block the simulation, then $Block_p$ is 0.

- Let $VCEF$ be the term identified earlier in this paper that captures the efficiency achieved by running at lower concurrency. E.g., if in-line visualization took 1 second on 10,000 nodes, but in-transit visualization took 5 seconds on 1,000 nodes, then $VCEF$ would be $\frac{1\times10,000}{5\times1,000}$, which is 2.

We have two terms for transferring data because sending data from the simulation side may be faster than receiving it on the in-transit side. For example, if 8 simulation nodes send to 1 visualization node, then that 1 visualization node will need to unserialize eight times as much data as each of the simulation nodes serialized.

– Let $Send_p$ be the proportion of time by the simulation code sending data to in-transit visualization resources. E.g., if T is 5 seconds and the send time is 2 seconds, then $Send_p = 2/5$, which is 0.4.

– Let $Recv_p$ be the proportion of time spent receiving data on the in-transit visualization resources. E.g., if T is 5 seconds and the transfer time is 2 seconds, then $Recv_p = 2/5$, which is 0.4.

7.3.2 Base Model Defined. We define our base cost model below. This cost model will be refined in Section 7.3.4 as we consider the implications of blocking. The cost for in-transit visualization will lower than in-line visualization when:

$$\begin{gathered}
\text{(total resources with in-transit)} \times \text{(time per cycle for simulation with in-transit)} \\
< \\
\text{(total resources with in-line)} \times \text{(time per cycle for simulation with in-line)} \\
\Longrightarrow \\
\text{(\# in-transit nodes + \# simulation nodes)} \times \\
\text{(simulation cycle time + transfer time + block time)} \\
< \\
\text{(\# simulation nodes)} \times \text{(simulation cycle time + in-line vis time)}
\end{gathered} \tag{7.1}$$

Using the terms defined above in Section 7.3.1, this becomes:

$$(N \times Res_p + N) \times (T + T \times Send_p + T \times Block_p) < (N) \times (T + T \times Vis_P) \tag{7.2}$$

This equation can be simplified by dividing both sides by the simulation cycle time (T) and number of nodes (N):

$$(1 + Res_p) \times (1 + Send_p + Block_p) < (1 + Vis_P) \tag{7.3}$$

If Equation 7.3 is true, then in-transit costs less than in-line.

7.3.3 Base Model Discussion. In-transit visualization has three different costs that do not occur with in-line. (1) In-transit visualization requires data transfer, which slows down the simulation nodes. (2) In-transit visualization requires dedicated resources beyond those required for in-line. If the in-transit visualization finishes quickly, these additional resources sit idle, and yet still incur cost. (3) In-transit can block the simulation if the visualization is not finished before the simulation is ready to send data for the next cycle. This is very harmful since it slows down the simulation nodes. There are alternatives to blocking, for example skipping cycles, and only visualizing the latest. In this study, our focus is on blocking, and we do not consider the alternatives.

Given the three additional costs incurred by in-transit, the *only* way for it to cost less than in-line is for the visualization to run faster at lower concurrency. In other words, the cost savings with in-transit can *only* occur if the benefit of ($VCEF$) outweighs the combined effects of the three additional costs described above. The fact that certain operations are more efficient at lower levels of concurrency provides an opporunity for a more cost effective solution.

That said, there are scenarios where any value of $VCEF$ is insufficient to achieve cost savings. Examples where in-transit can never be more cost effective, regardless of $VCEF$, are discussed below:

– If blocking takes longer than in-line visualization (e.g., $Block_p = 0.3, Vis_p = 0.2$), it is impossible to be more cost efficient. For example, even if $T = \epsilon$, then $(1 + \epsilon) \times (1 + \epsilon + 1.3) < (1 + 1.2)$ is not possible.
– Further, even if $Block_p = 0$ (no blocking), then some in-transit configurations will still always be less efficient:
    * if the simulation transfer cost is bigger than the in-line visualization time (e.g., $Send_p = 0.4, Vis_p = 0.2$), then: $(1 + \epsilon) \times (1 + 0.4 + 0) < 1.2$
    * if there are many in-transit nodes (e.g., $Res_p = 0.5$) and the in-line visualization time is sufficiently fast (e.g., $Vis_p = 0.5$), then: $(1 + 0.5) \times (1 + \epsilon + 0) < 1 + 0.5$

7.3.4 When Does Blocking Occur?: Replacing $Block_p$ via $VCEF$.

In this section we expand the model by using the $VCEF$ term to determine when blocking will occur. We then present two new equations that define when in-transit will cost less if blocking does or does not occur.

Consider what it means to block. Blocking occurs when in-transit resources are taking longer to do their job than the simulation resources are taking to do their job. Similarly, "not blocking" means that the in-transit resources are doing their job faster than the simulation resources take to do their job. So, what does "time to do their job" mean? For the simulation side, this means the time to advance the simulation plus the time to send the data, i.e., $T + T \times Send_p$. For the in-transit side, this means the time to receive data ($T \times Recv_p$) plus the time to do the visualization task. This latter time is explored below.

Nominally, assuming that visualization scaled perfectly as a function of concurrency, the cost (number of node seconds) to do the visualization task can be directly calculated from the in-line case: $N \times (Vis_p \times T)$. However, a key premise

of this study is that in-transit has an advantage at lower concurrency because of $VCEF$. Because in-transit is running at a lower concurrency, the cost is scaled by the $VCEF$ term: $\frac{N \times (Vis_p \times T)}{VCEF}$. Finally, the time to carry out the visualization task on the in-transit nodes would be the $VCEF$-reduced cost divided by the resources ($N \times Res_p$). Thus, the in-transit visualization time is:

$$\frac{N \times (T \times Vis_p)}{VCEF \times N \times Res_p} \tag{7.4}$$

Canceling out N gives a simpler form:

$$\frac{Vis_p \times T}{VCEF \times Res_p} \tag{7.5}$$

Restating, blocking occurs with in-transit when the time to receive data plus the visualization time is greater than the simulation time plus the time to send data:

$$Recv_p \times T + \frac{Vis_p \times T}{VCEF \times Res_p} > T \times (1 + Send_p) \tag{7.6}$$

This means that blocking *does not* occur if:

$$Recv_p \times T + \frac{Vis_p \times T}{VCEF \times Res_p} \leq T \times (1 + Send_p) \tag{7.7}$$

The terms in Equation 7.7 can be rearranged to find the $VCEF$ values when blocking *does not* occur:

$$\frac{Vis_p}{Res_p \times (1 + Send_p - Recv_P)} \leq VCEF \tag{7.8}$$

This analysis on blocking informs the original question: when does in-transit incur less cost than in-line? This can be answered using a combination of Equations 7.3 and our observations about blocking in this section. If blocking does not occur, then $Block_p$ drops out as zero, and Equation 7.3 is simplified:

$$(1 + Res_p) \times (1 + Send_p) < (1 + Vis_P) \tag{7.9}$$

If blocking does occur, then the simulation advances only as fast as the in-transit resources can take new data. This means that the time term for the left-hand side

of Equation 7.3, which was previously $1 + Send_p$, is replaced with the in-transit time. Using the relationship in Equation 7.6, we get:

$$(1 + Res_p) \times (Recv_p + \frac{Vis_p}{VCEF \times Res_p}) < (1 + Vis_P) \tag{7.10}$$

7.3.5 Cost Model Discussion. The basis of the cost model are described above in Equations 7.3, 7.8, 7.9, and 7.10. This model allows the relative costs of in-line and in-transit visualization for a particular configuration to be analyzed. The first step is to determine the cost feasibility of in-transit. Equation 7.3 serves as a threshold for determining when this is *possible*. If Equation 7.3 is false, in-line visualization is the cost-effective solution. Otherwise, when Equation 7.3 is true, Equations 7.8, 7.9, and 7.10 are used to determine cost feasibility based on blocking, as follows:

- The $VCEF$ value necessary to prevent blocking is given by Equation 7.8: $VCEF \geq \frac{Vis_p}{Res_p \times (1 + Send_p - Recv_P)}$
  - For cases when there is no blocking, using Equation 7.9 shows that in-transit is cost efficient if:

    $$(1 + Res_p) \times (1 + Send_p) < (1 + Vis_P) \text{ 2mm}$$

  - Otherwise, for cases where blocking occurs, using Equation 7.10 shows that in-transit is cost efficient if:

    $$(1 + Res_p) \times (Recv_p + \frac{Vis_p}{VCEF \times Res_p}) < (1 + Vis_P)$$

7.4 Results

In this section we use the model described in Section 7.3 to analyze the data collected from our experiments. In particular, we follow the discussion detailed in Section 7.3.5. In Section 7.4.1, we discuss and analyze the magnitude of $VCEF$

(Equation 7.8) for each experiment. In 7.4.2 we use Equation 7.3 from our model to determine the in-transit cost savings feasibility for each experiment. Finally, in Section 7.4.3, we combine these two and discuss the experiments that are feasible and have sufficient $VCEF$ to produce cost savings using in-transit for both non-blocking and blocking cases (Equations 7.9 and 7.10).

7.4.1 *VCEF* Magnitude Across Experiments. Figure 24 shows the $VCEF$ for each experiment. We felt the most surprising result was how large $VCEF$ values were as a whole. Many of the experiments had values above $4X$, which creates significant opportunities for the cost effectiveness of in-transit. Surprisingly, volume rendering experiments where the in-transit resources were 50% of the simulation ($Alloc(50\%)$) were able to achieve $VCEF$ values of about $4X$. Putting this number in perspective, if an $Alloc(50\%)$ experiment runs in the same amount of time as its in-line counterpart using half the concurrency, then its $VCEF$ would be 2. This is because it would have run using half the resources while taking the same amount of time as in-line. Higher values indicate that the runtime has decreased at smaller concurrency, i.e., 4X cost efficiency via using half the resources and running 2X faster. Further, we note this volume rendering algorithm has been extensively optimized and is used in a production setting. This result highlights the significant advantage that $VCEF$ provides. Algorithms with poor scalability (i.e., heavy communication) are able to run at lower levels of concurrency, and therefore achieve better performance.

As expected, $VCEF$ is heavily dependent on the type of algorithm. The volume rendering experiments were communication-heavy, lending itself to higher cost efficiency when running at lower concurrency. The isosurfacing experiments were computation-heavy — first, an isosurface is calculated, and then it was

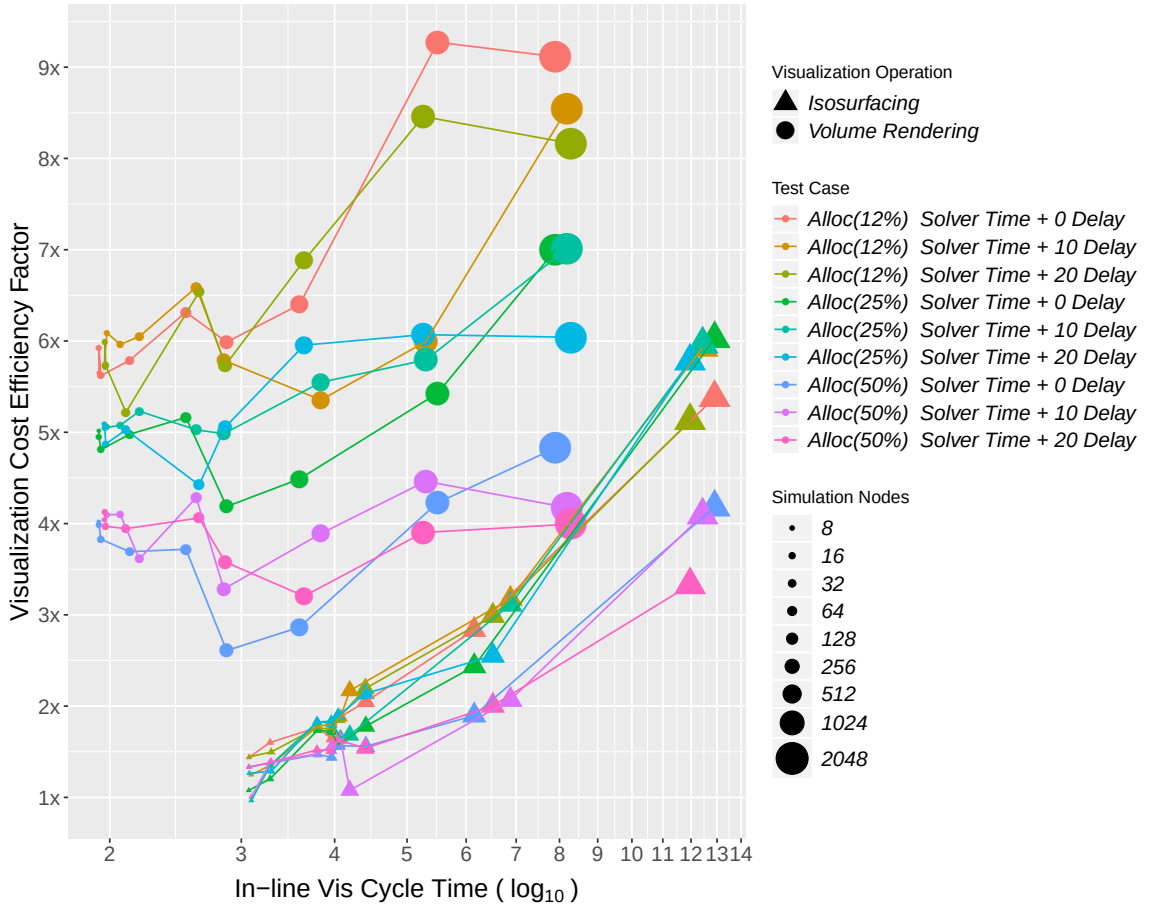


Figure 24. This plot shows in-transit $VCEF$ as a function of the in-line cycle time. Isosurfacing experiments are denoted with a triangle glyph and volume rendering with a circle glyph. Each glyph is scaled by the concurrency of the experiment (isosurfacing: 8-1024; volume rendering: 8-2048). Experiments are grouped by color (configuration) and connected by lines (concurrency sequence).

rendered. The isosurface calculation is embarrassingly parallel, so there is no reason to expect a high $VCEF$. That said, the parallel rendering became very slow at high concurrency, as evidenced by the high in-line times (>10 seconds). This was due to the communication required to perform the image compositing and the final reduction using the radix-k algorithm. In these cases, the $VCEF$ values increased from $3X$ to $6X$.

While the main takeaway of Figure 24 is high $VCEF$ values, a secondary takeaway looks ahead to our analysis of cost savings, and in particular establishing intuition about which configurations will be viable for cost savings. All volume rendering experiments had high $VCEF$ values, while only isosurfacing experiments at very high concurrency had high $VCEF$ values. The isosurfacing experiments at lower concurrencies had smaller $VCEF$ values, which makes them less likely to offset the additional costs incurred for in-transit (transfer times, blocking, idle).

7.4.2 Feasibility of Cost Savings. Equation 7.3 from our model is used to determine the feasibility of cost savings for in-transit visualization. When Equation 7.3 is true, then cost feasibility is possible. Figure 25 uses this equation to show the feasibility for each experiment. The black line shows where in-line and in-transit costs are identical, and the region above the black line is cost feasibility for in-transit. This figure follows discussion from Section 7.3.3. For example, if the in-line cost is less than the transfer cost, then no $VCEF$ value can make in-transit cost effective. Or if the resources devoted to in-transit are very large, then they will likely sit idle and be a incur cost at no gain. About half of our experiments were in this category, incapable of achieving cost savings with in-transit, because the transfer and resource costs exceeded the in-line costs. In the remaining half of the experiments, our choice for the number of in-transit nodes created a potentially feasible situation — the resources dedicated to in-transit and the cost of transferring data was less than the in-line visualization cost. That said, only some of these experiments actually led to cost savings with in-transit. This is because the feasibility test for Figure 25 placed no consideration on whether the in-transit resources were sufficient to perform the visualization task. In some cases, $VCEF$ was enough that the in-transit resources could complete its visualization

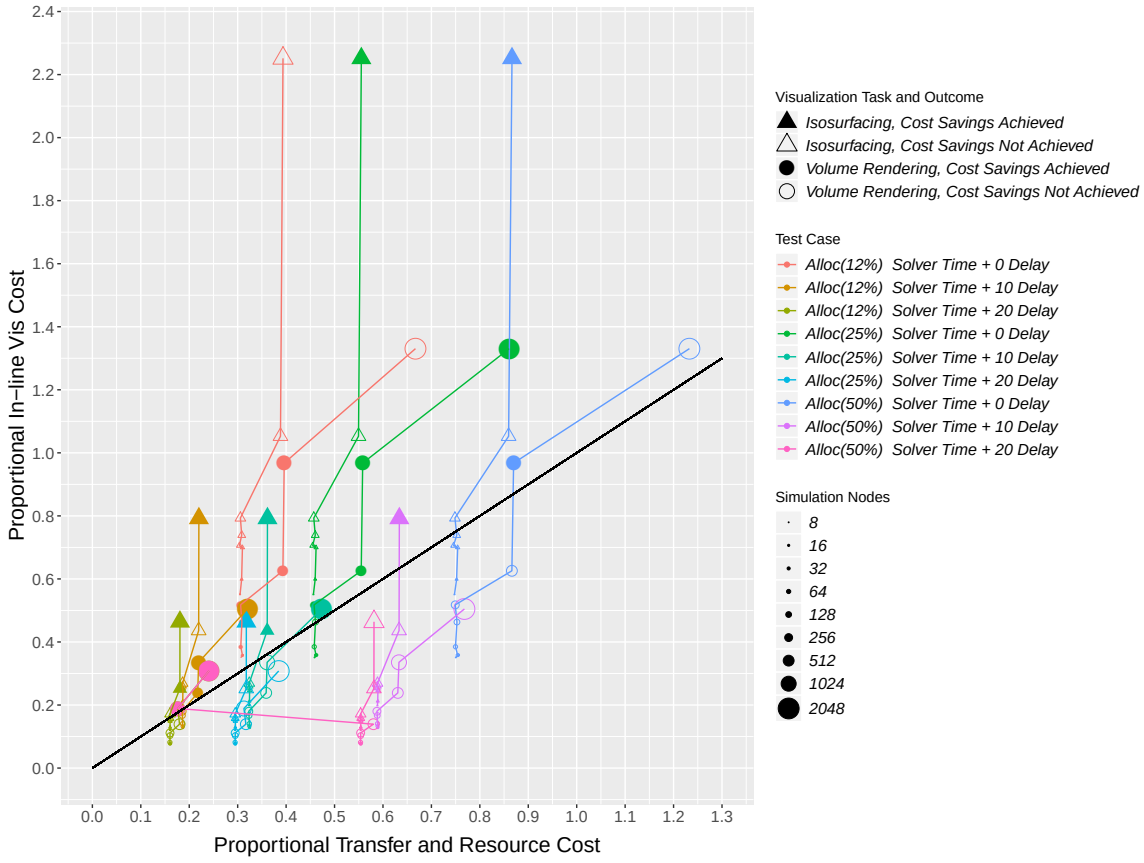


Figure 25. Plot of cost savings feasibility for each test case. Each glyph denotes the in-line cost as a function of transfer and resource costs. Glyph size represents the number of simulation nodes used in each test (isosurfacing: 8-1024; volume rendering: 8-2048). Hollow glyphs indicate in-line was more cost efficient and solid glyphs indicate that in-transit was more cost efficient. The black line marks where in-line and in-transit costs are equal. Above the line is where in-transit can be cost effective. In this plot, blocking is not considered. Some glyphs above the line are hollow however due to $VCEF$ being insufficient to achieve overall cost savings.

task within the allotted time. In others cases, $VCEF$ was not sufficient, and this caused the in-transit resources to block. Figure 26 takes this blocking into account, and faithfully plots the terms from Equation 7.3 from Section 7.3.2. The difference between Figure 25 and 26, then, is whether blocking is included when considering in-transit costs.

A final point from Figure 25 is the trend as concurrency increases — in-line visualization increases at a much higher rate than transfer costs. Consider the example of isosurfacing, with $Alloc(50\%)$ and $Delay(0)$ i.e., the blue lines on the right of Figure 25 with triangle glyphs. These experiments have in-line costs that go from 0.6X of the simulation cycle time at the smallest scale to 2.2X for the

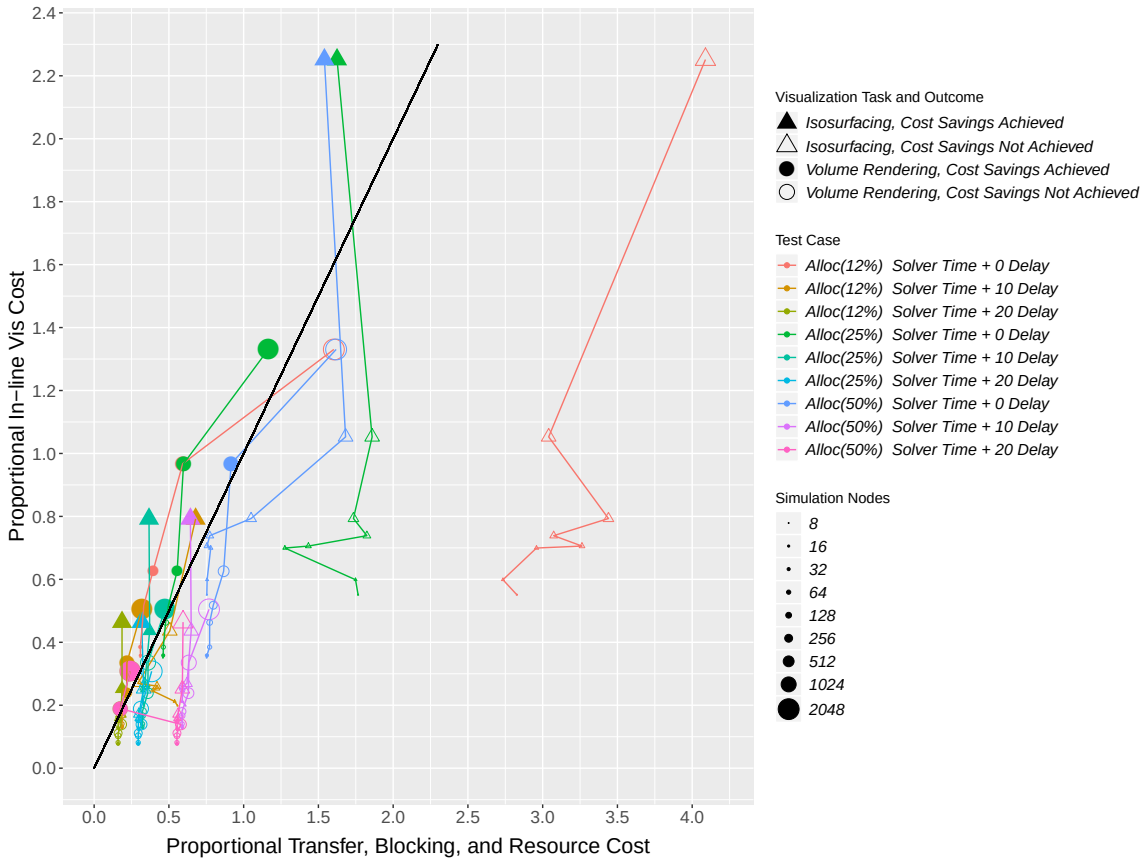


Figure 26. Plot of cost savings feasibility for each test case. Each glyph denotes the in-line cost as a function of transfer and resource costs. Glyph size represents the number of simulation nodes used in each test (isosurfacing: 8-1024; volume rendering: 8-2048). Hollow glyphs indicate in-line was more cost efficient and solid glyphs indicate that in-transit was more cost efficient. The black line marks where in-line and in-transit costs are equal. Above the line is where in-transit can be cost effective. This plot is an update of Figure 25 to include blocking costs. This plot demonstrates that our cost model is able to perfectly infer when cost savings can be achieved with in-transit, as only hollow glyphs appear below the black line and only solid glyphs appear above it.

largest scale. Further, the x-values (i.e., transfer cost and resource cost) change in a much more modest way (0.75X to 0.85X, with this representing only a variation in transfer since the resource cost is fixed at 0.5 for this case). This is a critical point to bring up for in-line visualization: It can be very difficult to scale some algorithms up to the scale of the simulation without incurring huge penalties. All of the other families of experiments exhibit a similar trend, with little variation in X (transfer and resource) and significant increases in Y (in-line visualization) as scale increases. Extrapolating forward, the opportunities demonstrated in our experiments will only become greater as supercomputers get larger and larger.

7.4.3 Achieved Cost Savings. Figure 27 extends Figure 26 by plotting the results of Equation 7.8 for each of the points that did provide cost savings. Equation 7.8 calculates the required $VCEF$ value for a in-transit experiment to not block the simulation. While blocking the simulation is certainly not an ideal configuration, it is still possible to achieve cost savings if the cost savings gained through $VCEF$ is greater than the cost of the blocked simulation. About a third of the experiments that provided cost savings from Figure 26 actually blocked the simulation (points to the right of the black line).

The main takeaway from this plot though, is the rate at which $VCEF$ allowed in-transit visualization to achieve cost savings and prevent blocking. About two thirds of the cases that achieved cost savings did so by not blocking the simulation. This was in large part due to the high values for $VCEF$ that were achieved in those cases.

Looking back to the intuition we established in Section 7.4.1 about which experiments would be viable from a cost savings standpoint, we see that our intuition was correct. Our intuition was that volume rendering would lead to

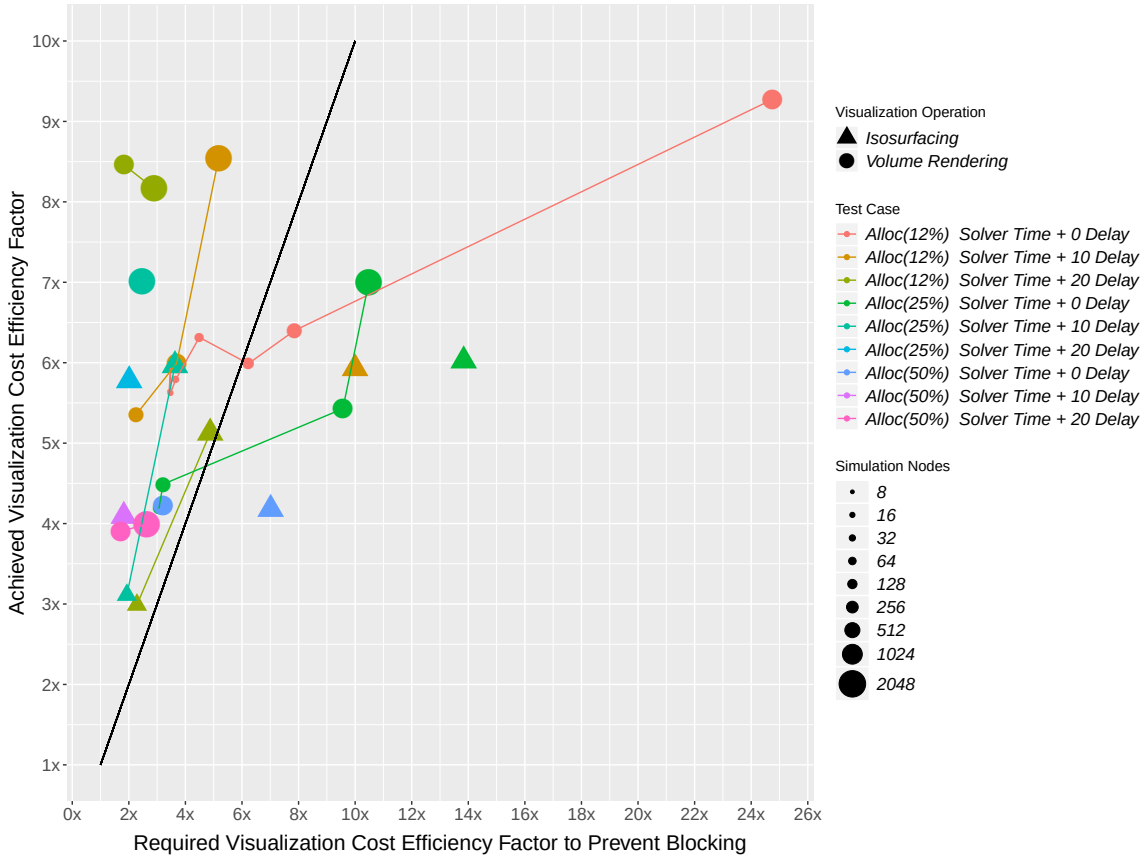


Figure 27. This plot takes the points from experiments in Figure 26 where in-transit was cost effective and plots the achieved $VCEF$ as a function of the required $VCEF$ to prevent blocking. The black line is Equation 7.8. Points above the line did not block, while those below did block. This plot shows two things: first, the necessary $VCEF$ speedup required to prevent blocking, and second, that cost feasibility is possible even with simulation blocking.

more experiments with cost savings vs. isosurfacing due to its high $VCEF$ values across all concurrencies, whereas isosurfacing only had high $VCEF$ values at high concurrency. Looking at Figure 27, we see that the majority of the points are for volume rendering, 19 cost winners, vs. isosurfacing, 9 cost winners. This trend indicates two important things: first, at even higher concurrency we should expect to see larger values for $VCEF$, with even more cost winners for in-transit, and second, in future as more algorithms are studied, those with even more

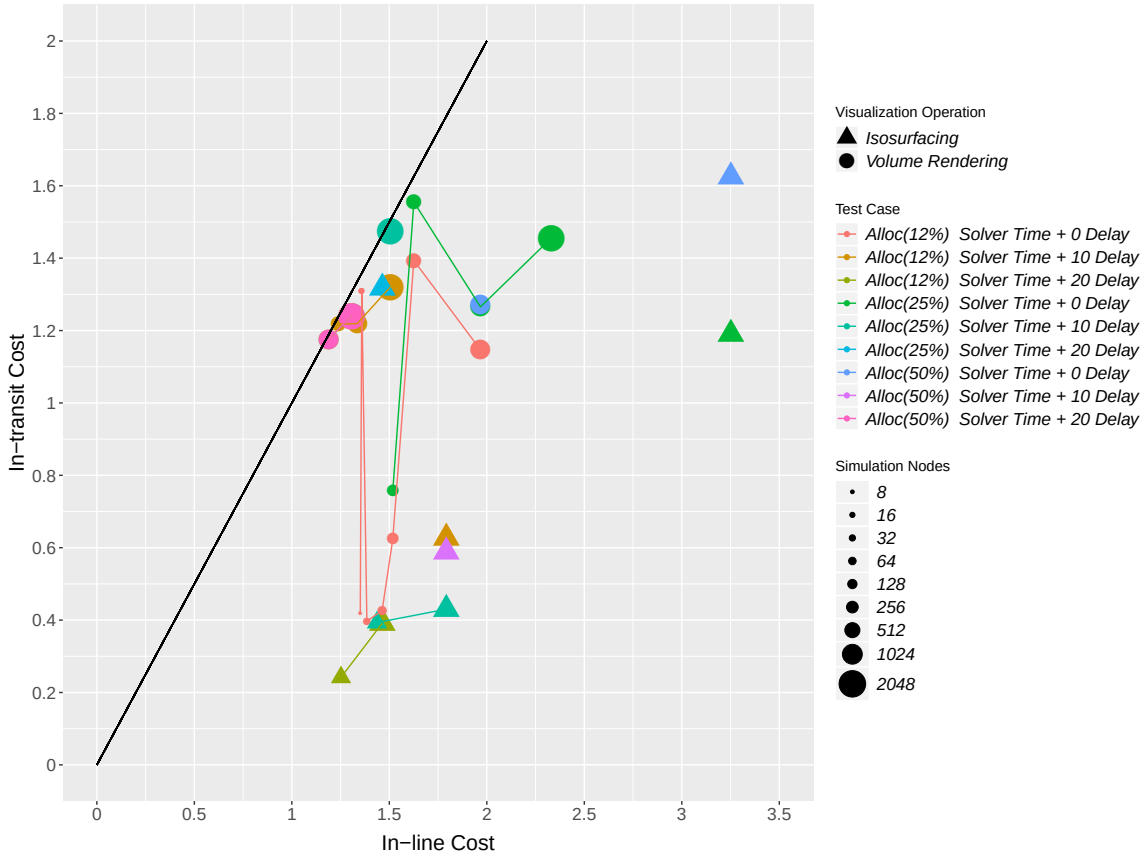


Figure 28. This plot takes the points from experiments in Figure 26 where in-transit was cost effective and plots the in-transit cost as a function of the in-line cost using Equation 7.9 (if no blocking occurred), or Equation 7.10 (otherwise). The black line indicates where costs are equal.

communication than volume rendering should see even greater cost savings due to $VCEF$.

Figure 28 takes all of the cases that achieved cost savings from Figure 26 and shows what the observed in-transit and in-line costs were in each case. The further the points are from the black line the larger the in-transit cost savings. This chart shows that 30 cases out of a possible 58 cases from Figure 25 were able to achieve cost savings. Meaning that overall, out of our 153 in-transit tests, we demonstrated high $VCEF$ values and cost savings in 30, or 20%, of our cases. We

note that these test cases were originally conceived for a study on the fastest time to solution, not cost savings, so seeing 20% of cases winning from a cost perspective is encouraging. Stated differently, our experiments did not focus on optimizing over resources, and so it is possible that more success could have been found. By focusing on smaller allocations, these studies should see a much higher percentage of cost winners for in-transit.

7.5 Summary

The primary results from this chapter are three-fold: (1) VCEF values are surprisingly high, and in particular high enough to create opportunities for in-transit to be cost effective over in-line, (2) a model for considering the relative costs between in-transit and in-line that incorporates $VCEF$, and (3) consideration of that model over a corpus of data that demonstrated that VCEF-based savings do in fact create real opportunities for in-transit cost savings. We feel this result is important, since it provides simulation teams a valuable metric to use in determining which in situ paradigm to select. Combined with in-transit's other benefits (such as fault tolerance), we feel this new information on cost could be impactful in making a decision. In our studies, our communication-heavy algorithm showed more promise for in-transit cost benefit than the computation-heavy algorithm. This observation speaks to an additional role for in-transit: sidestepping scalability issues by offering the ability to run at lower concurrency. This is particularly important as the visualization community considers critical algorithms like particle advection, topology, connected components, and Delaunay tetrahedralization. In terms of future work, we would like to explore $VCEF$ with more simulation codes and different algorithms, consider the implications to $VCEF$ if we choose to not perform visualization every step, what can be accomplished

if slack time on the in-transit nodes is used to perform additional visualization, and to explore the feasibility of creating models to predict $VCEF$ values for common visualization algorithms. Finally, we would like to incorporate a measure of uncertainty into our plots (and predictions once future work is completed) that accounts for system noise and variation in timings between different instances of the same test run. These additions will give end-users of our models and plot more realistic expectations for results for the set of tests that fall within these regions of uncertainty

CHAPTER VIII

CONCLUSIONS AND FUTURE DIRECTIONS

Some of the text in this chapter comes from [71, 73], which were described in detail in Chapter VI and Chapter VII.

8.1 Conclusions

In situ visualization for scientific simulations is becoming increasingly important as the discrepancy between compute and I/O continues to grow in modern supercomputers. In order for scientists to get the most knowledge out of their data they will need to embrace in situ methods for automatic analysis, visualization, reduction, and extract selection. Therefore, it is critical to understand how to perform these tasks efficiently, with the least impact to the simulation and the simulations cost. There is currently very limited work in the area of scalability performance understanding for in situ visualization techniques. This dissertation investigated the scalability and performance of two different common visualization algorithms from small to large scale both in-line and in-transit in order to answer its central question: *"In-line vs. in-transit insitu: which technique to use at scale?"*

In Section 1.2 we presented six sub-questions to aid in answering the central question of this dissertation. We will now look at those six sub-questions and what we discovered during the course of this dissertation's research.

Q: How does communication between ranks affect in-line visualization (is it more efficient for some algorithms vs. others)?

A: The idea that some algorithms are more efficient on smaller allocations due communication between the ranks is the core of this dissertation. We saw that the algorithm type does play an important role in determining what will be time and cost efficient. Isosurfacing has

very little communication, so ranks are free to operate on data almost independently. This is in stark contrast to volume rendering which has both a compute and communication component. We observed that the inter-rank communication with volume rendering (image compositing) was a primary driving factor in the high $VCEF$ values for all of the isosurfacing test cases. These results suggest then that for communication heavy algorithms at large scale, that in-line visualization is a poor choice if done frequently, and will incur large costs for the simulation vs. in-transit. On the other hand for computation heavy algorithms, in-line performed well at moderate scale, only failing at the scaling limits of our study.

Q: At lower concurrency, are in-line techniques more efficient?

A: This depends highly on the cycle time of the simulation. In-line benefits less from longer cycle times than in-transit visualization, so it has fewer opportunities for wins. With shorter simulation cycles however, there are cases where in-line visualization at lower concurrencies is more efficient than in-transit. This stems in great part from a delicate balance between the compute and network requirements for visualization operations. At smaller concurrencies, these algorithms may no longer be network bound, so are able to efficiently take advantage of the in-line resources, being more efficient than the corresponding, even smaller, in-transit allocations.

Q: What are the overheads associated with in-transit techniques?

A: In-transit techniques can have fairly significant overhead in terms of cost. This is due to the required additional resources that in-transit requires for operation, as well as the time to complete the desired visualization tasks. These costs can be tailored, however, to a simulation's budget by modifying the frequency of visualization, the complexity of the visualizations performed, and size of the visualization allocation. **In this way the cost of in-transit visualization can in fact become a net positive for the simulation by moving some inherently unscalable visualization algorithms off of the simulation resources to smaller visualization allocations.**

Q: Does in-transit ever cost less to use than in-line?

A: Yes, in-transit does in fact cost less than in-line in certain configurations. In our studies we saw a cost savings for in-transit in 30 cases, or 20% of the time. With continued understanding of the performance of visualization algorithms we can expect this percentage to rise significantly.

Q: What percentage of simulation resources are needed for in-transit so that it does not block the simulation (so that it keeps up)?

A: This question depends upon the visualization algorithm chosen, the simulation cycle time, as well as the visualization frequency. If the simulation cycle time is long enough, then it is possible to use 12% of the simulation resources or less and not block the simulation using in-transit visualization. As the simulation cycle decreases that percentage will have to rise, or the visualization frequency will need to drop.

Q: What size of resource allocation is needed for in-transit visualization so that resources are not wasted when doing infrequent visualization?

A: This question depends upon the same factors as the previous question. That is, the longer the simulation cycle time the fewer resources are needed in order to keep up with the simulation. For example, in our study, when we had the 15 and 25 second simulation cycle times, we needed fewer than $Alloc(12\%)$ resources for most cases, meaning that in order to not waste resources we needed to test smaller allocations. The shorter the cycle time the less impact we see from wasting in-transit resources as they are completely busy in most all cases.

Summarizing the findings of this dissertation, we found that the type of visualization algorithm is critical in deciding where to place the operation, either in-line or in-transit. With high communication algorithms we see that in-transit visualization has some of the best prospects at scale. Whereas, low communication algorithms more favor in-line visualization at scale.

A primary contribution of our work is the identification of $VCEF$ and the associated model that was developed for considering relative costs between in-line and in-transit visualization. We feel this result is important, since it provides simulation teams a valuable metric to use in determining which in situ paradigm to select. Combined with in-transit's other benefits (such as fault tolerance), we feel this new information on cost could be impactful in making a decision on placement. In our studies, our communication-heavy algorithm showed more promise for in-transit cost benefit than the computation-heavy algorithm. **This observation speaks to an additional role for in-transit: sidestepping scalability issues by offering the ability to run at lower concurrencies.**

8.2 Future Work

In the following subsections, we detail five areas of future research that build on the work from this dissertation.

8.2.1 Selecting Appropriately Sized In-Transit Allocations.

The first direction is in selecting an in-transit allocation that is likely to create cost benefits. Our corpus of data was originally conceived for a study on time savings. This is why it included configurations like $Alloc(50\%)$, which have very little chance of providing cost savings. Saying it another way, although we put little effort into choosing configurations that could achieve cost savings, we still found these cost savings occurred 20% of the time. If we put more effort into choosing such configurations, perhaps by incorporating the work of Malakar [89, 90], who had complementary ideas on choosing allocation sizes and analysis frequencies, this proportion could rise significantly. A twin benefit to choosing an appropriately sized in-transit allocation is that potentially more nodes would be available for simulation use, as over allocating an in-transit allocation can limit the maximum size of a simulation scaling run.

8.2.2 Understanding the Benefits of Hybrid In Situ Methods.

The second direction is in exploring the benefits and applications of hybrid in situ methods. Hybrid in situ methods have the potential to remove the negatives of both the in-line and in-transit paradigms. That is, for algorithms that are embarrassingly parallel and operate most efficiently at simulation scale, it would be faster, and most likely, more cost effective, to perform the algorithm on the simulation resources. Algorithms that are communication-bound perform most efficiently at scales below that of the full simulation. In this latter case, moving the data to the in-transit allocation would be the best choice.

Combining these two ideas, it is possible to create a system of in-line and in-transit resources that cooperates to achieve the best performance for the lowest cost. Take for example the first algorithm that we presented in Chapter VI, isocontouring plus rendering. With this algorithm it would be most efficient to first perform the isocontouring on the in-line resources and then move the intermediate result to a in-transit allocation. Then, the in-transit allocation can perform the parallel rendering and compositing at a smaller scale, being much more efficient.

The research here comes in on two fronts. First, it is not clear which algorithms can be performed in a hybrid environment. In fact, some algorithms may need to be rewritten to break them into separate phases that can each be called on different resources. Second, the time and cost savings with this form of in situ is currently based off of experience and conjecture, studies to determine these for hybrid in situ are still required. Furthermore, it is likely that hybrid methods will see a drop in the necessary in-transit resources needed to both keep up with the simulation and to stay cost efficient, provided that high computation algorithms are performed on the simulation resources.

8.2.3 Understanding and Predicting *VCEF*. The third direction is in understanding and being able to predict $VCEF$. For our study, we ran production software for two algorithms. We were able to observe $VCEF$ factors after the run, but we are not able to predict them. Predicting $VCEF$ is hard — it will vary based on algorithm, data size, architecture, and possibly due to data-dependent factors. This difficulty may even increase when hybrid in situ is considered, as there are more variables and costs to consider. More studies will need to be performed on a wider set of algorithms, data sizes and complexities, and in-depth AI models created to model each algorithm with an understanding

of different machine architectures. These studies however will have great benefit, as being able to predict $VCEF$ would lead to being able to choose cost effective configurations for in situ visualization routines.

8.2.4 Alternatives to Blocking the Simulation and Idle In-Transit Resources. The fourth direction is in considering more alternatives to blocking the simulation and having in-transit resource sitting idle. Making the choice to block simplified our cost model and study. A twin choice was to ignore idle time — we could have tried to do "more visualization" when the in-transit resources completed their initial task and went idle. Making a system that is more dynamic (not blocking and instead visualizing data from the next time step and/or also adding tasks when there is idle time) would be an interesting future direction. Such a system would be able to realize cost savings compared to in-line, provided $VCEF$ can offset transfer costs.

8.2.5 Incorporating a *VCEF* Predictor into a Visualization Workflow Generator. The fifth direction is in extending our understanding of $VCEF$ once we are able to predict it, and incorporate this into scientific workflows in a natural and agile way. For instance, a simulation team may have a set of 10 different intentions, or analysis operations, that they would like completed during a simulation campaign. Each of these intentions has a need for a certain temporal fidelity, data fidelity, accuracy, frequency, and timeliness. In addition, each intention will have a priority associated with it, that indicates if it is an operation that is critical and must happen, or if it would just be nice to have if there is available time and the cost is low enough. By having a model for predicting $VCEF$, a semantic system like this can be created that will allow for easy creation, scheduling, and cost efficient use of analysis and visualization with a simulation.

A critical component to the success of this system will be in the use of learning techniques that will allow the $VCEF$ predictor and the overall workflow predictor to learn on the fly, enabling it to make better predictions and scheduling choices as machines, workloads, data sizes, and data density changes. An analysis workflow tool like this will help to make visualization and analysis better, more prevalent, and cheaper, helping to push forward scientific discovery.

REFERENCES CITED

[1] Cactus. .http://cactuscode.org/. Accessed: 2016-11-27.

[2] Cloverleaf3d. http://uk-mac.github.io/CloverLeaf3D/. Accessed: 2018-3-1.

[3] Conduit. https://www.github.com/LLNL/conduit. Accessed: 2016-11-27.

[4] Damaris. http://damaris.gforge.inria.fr/. Accessed: 2016-11-27.

[5] Damaris/viz. http://damaris.gforge.inria.fr/doku.php. Accessed: 2016-11-27.

[6] Epsn. http://www.labri.fr/projet/epsn/. Accessed: 2016-11-27.

[7] Freprocessing. https://www.github.com/tfogal/freeprocessing. Accessed: 2016-11-27.

[8] The mesa 3d graphics library. http://www.mesa3d.org/. Accessed: 2016-11-27.

[9] Top 500d. https://www.top500.org/lists/2016/06/. Accessed: 2016-10-25.

[10] Hasan Abbasi, Matthew Wolf, Greg Eisenhauer, Scott Klasky, Karsten Schwan, and Fang Zheng. Datastager: scalable data staging services for petascale applications. *Cluster Computing*, 13(3):277–290, 2010.

[11] Alexy Agranovsky, David Camp, Christoph Garth, E Wes Bethel, Kenneth I Joy, and Hank Childs. Improved post hoc flow analysis via lagrangian representations. In *Large Data Analysis and Visualization (LDAV), 2014 IEEE 4th Symposium on*, pages 67–75. IEEE, 2014.

[12] Sean Ahern, Arie Shoshani, Kwan-Liu Ma, Alok Choudhary, Terence Critchlow, Scott Klasky, Valerio Pascucci, J Ahrens, EW Bethel, and H Childs. Scientific discovery at the exascale. *Report from the DOE ASCR 2011 Workshop on Exascale Data Management*, 2011.

[13] James Ahrens, Berk Geveci, and Charles Law. Visualization in the paraview framework. In Chuck Hansen and Chris Johnson, editors, *The Visualization Handbook*, pages 162–170, 2005.

[14] James Ahrens, Bruce Hendrickson, Gabrielle Long, Steve Miller, Rob Ross, and Dean Williams. Data-intensive science in the us doe: case studies and future challenges. *Computing in Science & Engineering*, 13(6):14–24, 2011.

[15] James Ahrens, Sébastien Jourdain, Patrick O'Leary, John Patchett, David H Rogers, and Mark Petersen. An image-based approach to extreme scale in situ visualization and analysis. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 424–434. IEEE Press, 2014.

[16] James Ahrens, John Patchett, Andrew Bauer, Sébastien Jourdain, David H Rogers, Mark Petersen, Benjamin Boeckel, Patrick O'Leary, Patricia Fasel, and Francesca Samsel. In situ mpas-ocean image-based visualization. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, Visualization & Data Analytics Showcase*, 2014.

[17] G Aloisioa, S Fiorea, Ian Foster, and D Williams. Scientific big data analytics challenges at large scale. *Proceedings of Big Data and Extreme-scale Computing (BDEC)*, 2013.

[18] Utkarsh Ayachit, Andrew Bauer, Earl P. N. Duque, Greg Eisenhauer, Nicola Ferrier, Junmin Gu, Kenneth Jansen, Burlen Loring, Zarija Lukić, Suresh Menon, Dmitriy Morozov, Patrick O'Leary, Michel Rasquin, Christopher P. Stone, Venkat Vishwanath, Gunther H. Weber, Brad Whitlock, Matthew Wolf, K. John Wu, and E. Wes Bethel. Performance Analysis, Design Considerations, and Applications of Extreme-scale *In Situ* Infrastructures. In *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC16)*, Salt Lake City, UT, USA, November 2016. LBNL-1007264.

[19] Utkarsh Ayachit, Andrew Bauer, Berk Geveci, Patrick O'Leary, Kenneth Moreland, Nathan Fabian, and Jeffrey Mauldin. Paraview catalyst: Enabling in situ data analysis and visualization. In *Proceedings of the First Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization*, pages 25–29. ACM, 2015.

[20] Andrew C Bauer, Hasan Abbasi, James Ahrens, Hank Childs, Berk Geveci, Scott Klasky, Kenneth Moreland, Patrick O'Leary, Venkatram Vishwanath, Brad Whitlock, and Wes Bethel. In situ methods, infrastructures, and applications on high performance computing platforms. In *Computer Graphics Forum*, volume 35, pages 577–597. Wiley Online Library, 2016.

[21] Nathan Bell and Jared Hoberock. Thrust: A productivity-oriented library for cuda. *GPU computing gems Jade edition*, 2:359–371, 2011.

[22] Janine C. Bennett, Hasan Abbasi, Peer-Timo Bremer, Ray Grout, Attila Gyulassy, Tong Jin, Scott Klasky, Hemanth Kolla, Manish Parashar, Valerio Pascucci, Philippe Pebay, David Thompson, Hongfeng Yu, Fan Zhang, and Jacqueline Chen. Combining in-situ and in-transit processing to enable extreme-scale scientific analysis. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 49:1–49:9, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[23] Keren Bergman, Shekhar Borkar, Dan Campbell, William Carlson, William Dally, Monty Denneau, Paul Franzon, William Harrod, Kerry Hill, Jon Hiller, et al. Exascale computing study: Technology challenges in achieving exascale systems. *Defense Advanced Research Projects Agency Information Processing Techniques Office (DARPA IPTO), Tech. Rep*, 15, 2008.

[24] Wes Bethel. Visapult: A prototype remote and distributed visualization application and framework. *Lawrence Berkeley National Laboratory*, 2000.

[25] Wes Bethel. Visualization dot com. *Computer Graphics and Applications, IEEE*, 20(3):17–20, 2000.

[26] Wes Bethel, Cristina Siegerist, John Shalf, Praveenkumar Shetty, TJ Jankun-Kelly, Oliver Kreylos, and Kwan-Liu Ma. Visportal: Deploying grid-enabled visualization tools through a web-portal interface. *Lawrence Berkeley National Laboratory*, 2003.

[27] A Bishop and P Messina. Scientific grand challenges in national security: The role of computing at the extreme scale. In *ASCR Scientific Grand Challenges Workshop Series, Tech. Rep*, 2009.

[28] Roger Blandford, Young-Kee Kim, and Norman Christ. Challenges for the understanding the quantum universe and the role of computing at the extreme scale. In *ASCR Scientific Grand Challenges Workshop Series, Tech. Rep*, 2008.

[29] Guy E Blelloch. *Vector models for data-parallel computing*, volume 356. MIT press Cambridge, 1990.

[30] David A Boyuka, Sriram Lakshminarasimham, Xiaocheng Zou, Zhenhuan Gong, John Jenkins, Eric R Schendel, Norbert Podhorszki, Qing Liu, Scott Klasky, and Nagiza F Samatova. Transparent in situ data transformations in adios. In *Cluster, Cloud and Grid Computing (CCGrid), 2014 14th IEEE/ACM International Symposium on*, pages 256–266. IEEE, 2014.

[31] Nick Brown, Rupert Nash, Gordon Gibb, Bianca Prodan, Max Kontak, Vyacheslav Olshevsky, and Wei Der Chien. The role of interactive super-computing in using hpc for urgent decision making. In *International Conference on High Performance Computing*, pages 528–540. Springer, 2019.

[32] Marc Buffat, Anne Cadiou, Lionel Le Penven, and Christophe Pera. In situ analysis and visualization of massively parallel computations. *International Journal of High Performance Computing Applications*, page 1094342015597081, 2015.

[33] CS Chang, S Ku, PH Diamond, Z Lin, S Parker, TS Hahm, and N Samatova. Compressed ion temperature gradient turbulence in diverted tokamak edge. *Physics of Plasmas (1994-present)*, 16(5):056108, 2009.

[34] Jerry Chen, Ilmi Yoon, and Wes Bethel. Interactive, internet delivery of visualization via structured prerendered multiresolution imagery. *IEEE Transactions on Visualization and Computer Graphics*, 14(2):302–312, 2008.

[35] Hank Childs, Eric Brugger, Kathleen Bonnell, Jeremy Meredith, Mark Miller, Brad Whitlock, and Nelson Max. A contract based system for large data visualization. In *VIS 05. IEEE Visualization, 2005.*, pages 191–198. IEEE, 2005.

[36] Hank Childs, Eric Brugger, Brad Whitlock, Jeremy Meredith, Sean Ahern, David Pugmire, Kathleen Biagas, Mark Miller, Cyrus Harrison, Gunther H. Weber, Hari Krishnan, Thomas Fogal, Allen Sanderson, Christoph Garth, E. Wes Bethel, David Camp, Oliver Rübel, Marc Durant, Jean M. Favre, and Paul Navrátil. VisIt: An End-User Tool For Visualizing and Analyzing Very Large Data. In *High Performance Visualization–Enabling Extreme-Scale Scientific Insight*, pages 357–372. Chapman and Hall/CRC, Oct 2012.

[37] Hank Childs, Berk Geveci, Will Schroeder, Jeremy Meredith, Kenneth Moreland, Christopher Sewell, Torsten Kuhlen, and E Wes Bethel. Research challenges for visualization software. *Computer*, 46(5):34–42, 2013.

[38] Hank Childs, Ma Kwan-Liu, Yu Hongfeng, Whitlock Brad, Meredith Jeremy, Favre Jean, Klasky Scott, Podhorszki Norbert, Schwan Karsten, Wolf Matthew, Parashar Manish, and Zhang Fan. In situ processing. In E. Wes Bethel, Hank Childs, and Charles Hansen, editors, *High Performance Visualization: Enabling Extreme-Scale Scientific Insight*. CRC Press, Boca Raton, FL, 2012.

[39] Hank Childs, David Pugmire, Sean Ahern, Brad Whitlock, Mark Howison, Prabhat, Gunther H. Weber, and E. Wes Bethel. Extreme scaling of production visualization software on diverse architectures. *IEEE Comput. Graph. Appl.*, 30(3):22–31, May 2010.

[40] Hank Childs, David Pugmire, Sean Ahern, Brad Whitlock, Mark Howison, Prabhat, Gunther H. Weber, and E. Wes Bethel. Visualization at extreme scale concurrency. In E. Wes Bethel, Hank Childs, and Charles Hansen, editors, *High Performance Visualization: Enabling Extreme-Scale Scientific Insight*. CRC Press, Boca Raton, FL, 2012.

[41] Jong Y Choi, Kesheng Wu, Jacky C Wu, Alex Sim, Qing G Liu, Matthew Wolf, C Chang, and Scott Klasky. Icee: Wide-area in transit data processing framework for near real-time scientific applications. In *4th SC Workshop on Petascale (Big) Data Analytics: Challenges and Opportunities in conjunction with SC13*, 2013.

[42] Jai Dayal, Drew Bratcher, Greg Eisenhauer, Karsten Schwan, Matthew Wolf, Xuechen Zhang, Hasan Abbasi, Scott Klasky, and Norbert Podhorszki. Flexpath: Type-based publish/subscribe system for large-scale science analytics. In *14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing(CCGRID '14)*, 2014.

[43] Ciprian Docan, Manish Parashar, and Scott Klasky. Dataspaces: an interaction and coordination framework for coupled simulation workflows. *Cluster Computing*, 15(2):163–181, 2012.

[44] Ciprian Docan, Fan Zhang, Tong Jin, Hoang Bui, Qian Sun, Julian Cummings, Norbert Podhorszki, Scott Klasky, and Manish Parashar. Activespaces: Exploring dynamic code deployment for extreme scale data processing. *Concurrency and Computation: Practice and Experience*, pages 1–22, 2014.

[45] Matthieu Dorier, Roberto Sisneros, Tom Peterka, Gabriel Antoniu, and Dave Semeraro. Damaris/viz: A nonintrusive, adaptable and user-friendly in situ visualization framework. In *LDAV-IEEE Symposium on Large-Scale Data Analysis and Visualization*, 2013.

[46] Earl P Duque, Daniel E Hiepler, Robert Haimes, Christopher P Stone, Steven E Gorrell, Matthew Jones, and Ron Spencer. Epic–an extract plug-in components toolkit for in situ data extracts architecture. In *22nd AIAA Computational Fluid Dynamics Conference*, page 3410, 2015.

[47] Stefan Eilemann and Renato Pajarola. Direct send compositing for parallel sort-last rendering. In *Proceedings of the 7th Eurographics conference on Parallel Graphics and Visualization*, pages 29–36. Eurographics Association, 2007.

[48] David Ellsworth, Bryan Green, Chris Henze, Patrick Moran, and Timothy Sandstrom. Concurrent visualization in a production supercomputing environment. *Visualization and Computer Graphics, IEEE Transactions on*, 12(5):997–1004, 2006.

[49] T Todd Elvins. A survey of algorithms for volume visualization. *ACM Siggraph Computer Graphics*, 26(3):194–201, 1992.

[50] Nathan Fabian, Kenneth Moreland, David Thompson, Andrew Bauer, Pat Marion, Berk Geveci, Michel Rasquin, and Kenneth Jansen. The paraview coprocessing library: A scalable, general purpose in situ visualization library. In *Large Data Analysis and Visualization (LDAV), 2011 IEEE Symposium on*, pages 89–96. IEEE, 2011.

[51] Oliver Fernandes, David S Blom, Steffen Frey, Alexander H Van Zuijlen, Hester Bijl, and Thomas Ertl. On in-situ visualization for strongly coupled partitioned fluid-structure interaction. In *Coupled Problems 2015: Proceedings of the 6th International Conference on Computational Methods for Coupled Problems in Science and Engineering, Venice, Italy, 18-20 May 2015*. CIMNE, 2015.

[52] Thomas Fogal, Fabian Proch, Alexander Schiewe, Olaf Hasemann, Andreas Kempf, and Jens Krüger. Freeprocessing: Transparent in situ visualization via data interception. In *Eurographics Symposium on Parallel Graphics and Visualization: EG PGV:[proceedings]/sponsored by Eurographics Association in cooperation with ACM SIGGRAPH. Eurographics Symposium on Parallel Graphics and Visualization*, volume 2014, page 49. NIH Public Access, 2014.

[53] Brian Friesen et al. In situ and in-transit analysis of cosmological simulations. *Computational Astrophysics and Cosmology*, 3(1):4, 2016.

[54] Guilia Galli and Thom Dunning. Discovery in basic energy sciences: The role of computing at the extreme scale. In *ASCR Scientific Grand Challenges Workshop Series, Tech. Rep*, 2009.

[55] G. Gibb, R. Nash, N. Brown, and B. Prodan. The technologies required for fusing hpc and real-time data to support urgent computing. In *2019 IEEE/ACM HPC for Urgent Decision Making (UrgentHPC)*, pages 24–34, Nov 2019.

[56] Tom Goodale, Gabrielle Allen, Gerd Lanfermann, Joan Massó, Thomas Radke, Edward Seidel, and John Shalf. The cactus framework and toolkit: Design and applications. In *International Conference on High Performance Computing for Computational Science*, pages 197–227. Springer, 2002.

[57] Junmin Gu, Burlen Loring, Kesheng Wu, and E Wes Bethel. Hdf5 as a vehicle for in transit data movement. In *Proceedings of the Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization*, pages 39–43, 2019.

[58] Shinichi Habata, Mitsuo Yokokawa, Shigemune Kitawaki, et al. The earth simulator system. *NEC Research and Development*, 44(1):21–26, 2003.

[59] Salman Habib, Robert Roser, Richard Gerber, Katie Antypas, Katherine Riley, Tim Williams, Jack Wells, Tjerk Straatsma, A Almgren, J Amundson, et al. Ascr/hep exascale requirements review report. *arXiv preprint arXiv:1603.09303*, 2016.

[60] R Haimes. pv3. `3http://raphael.mit.edu/pv3/`. Accessed: 2016-11-27.

[61] Robert Haimes. pv3-a distributed system for large-scale unsteady cfd visualization. In *32nd Aerospace Sciences Meeting and Exhibit*, page 321, 1994.

[62] Randy Heiland and M Pauline Baker. A survey of co-processing systems. *CEWES MSRC PET Technical Report*, pages 98–52, 1998.

[63] Seif Ibrahim, Thomas Stitt, Matthew Larsen, and Cyrus Harrison. Interactive in situ visualization and analysis using ascent and jupyter. In *Proceedings of the Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization*, pages 44–48, 2019.

[64] Christopher Johnson, Steven G Parker, Charles Hansen, Gordon L Kindlmann, and Yarden Livnat. Interactive simulation and visualization. *Computer*, 32(12):59–65, 1999.

[65] Wesley Kendall, Tom Peterka, Jian Huang, Han-Wei Shen, and Robert Ross. Accelerating and benchmarking radix-k image compositing at large scale. In *Proceedings of the 10th Eurographics conference on Parallel Graphics and Visualization*, pages 101–110. Eurographics Association, 2010.

[66] Scott Klasky, Hasan Abbasi, Jeremy Logan, Manish Parashar, Karsten Schwan, Arie Shoshani, Matthew Wolf, Sean Ahern, Ilkay Altintas, Wes Bethel, Luis Chacon, CS Chang, Jackie Chen, Hank Childs, Julian Cummings, Stephane Etheir, Ray Grout, Zhihong Lin, Quing Liu, Xiaosong Ma, Kenneth Moreland, Valerio Pascucci, John Wu, and Weikun Yu. In situ data processing for extreme-scale computing. *Scientific Discovery through Advanced Computing Program (SciDAC'11)*, 2011.

[67] J Kohl. Cumulvs. `http://www.csm.ornl.gov/cs/cumulvs.html`. Accessed: 2016-11-27.

[68] M. Kontak, J. Vidal, and J. Tierny. Statistical parameter selection for clustering persistence diagrams. In *2019 IEEE/ACM HPC for Urgent Decision Making (UrgentHPC)*, pages 7–12, Nov 2019.

[69] James Kress, Randy Michael Churchill, Scott Klasky, Mark Kim, Hank Childs, and David Pugmire. Preparing for in situ processing on upcoming leading-edge supercomputers. *Supercomputing Frontiers and Innovations*, 3(4):49–65, 2016.

[70] James Kress, Scott Klasky, Norbert Podhorszki, Jong Choi, Hank Childs, and David Pugmire. Loosely coupled in situ visualization: A perspective on why it's here to stay. In *Proceedings of the First Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization (ISAV), held in conjunction with SC15*, pages 1–6, Austin, TX, November 2015.

[71] James Kress, Matthew Larsen, Jong Choi, Mark Kim, Matthew Wolf, Norbert Podhorszki, Scott Klasky, Hank Childs, and David Pugmire. Comparing the time-to-solution for in situ visualization paradigms at scale.

[72] James Kress, Matthew Larsen, Jong Choi, Mark Kim, Matthew Wolf, Norbert Podhorszki, Scott Klasky, Hank Childs, and David Pugmire. Comparing the efficiency of in situ visualization paradigms at scale. In *ISC High Performance 2019*. ISC, 2019.

[73] James Kress, Matthew Larsen, Jong Choi, Mark Kim, Matthew Wolf, Norbert Podhorszki, Scott Klasky, Hank Childs, and David Pugmire. Opportunities for cost savings with in-transit visualization. In *ISC High Performance 2020*. ISC, 2020.

[74] James Kress, David Pugmire, Scott Klasky, and Hank Childs. Visualization and analysis requirements for in situ processing for a large-scale fusion simulation code. In *Proceedings of the 2nd Workshop on In Situ Infrastructures for Enabling Extreme-scale Analysis and Visualization*, pages 45–50. IEEE Press, 2016.

[75] T Kuhlen, R Pajarola, and K Zhou. 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)*, 2011.

[76] Sriram Lakshminarasimhan, Neil Shah, Stephane Ethier, Scott Klasky, Rob Latham, Rob Ross, and Nagiza F Samatova. Compressing the incompressible with isabela: In-situ reduction of spatio-temporal data. In *European Conference on Parallel Processing*, pages 366–379. Springer, 2011.

[77] Matthew Larsen, James Ahrens, Utkarsh Ayachit, Eric Brugger, Hank Childs, Berk Geveci, and Cyrus 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*, pages 42–46. ACM, 2017.

[78] Matthew Larsen, Eric Brugger, Hank Childs, Jim Eliot, Kevin Griffin, and Cyrus Harrison. Strawman: A batch in situ visualization and analysis infrastructure for multi-physics simulation codes. In *Proceedings of the First Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization*, pages 30–35. ACM, 2015.

[79] Matthew Larsen, Amy Woods, Nicole Marsaglia, Ayan Biswas, Soumya Dutta, Cyrus Harrison, and Hank Childs. A flexible system for in situ triggers. In *Proceedings of the Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization*, pages 1–6, 2018.

[80] Shaomeng Li, Kenny Gruchalla, Kristin Potter, John Clyne, and Hank Childs. Evaluating the Efficacy of Wavelet Configurations on Turbulent-Flow Data. In *Proceedings of IEEE Symposium on Large Data Analysis and Visualization (LDAV)*, pages 81–89, Chicago, IL, October 2015.

[81] Ning Liu, Jason Cope, Philip Carns, Christopher Carothers, Robert Ross, Gary Grider, Adam Crume, and Carlos Maltzahn. On the role of burst buffers in leadership-class storage systems. In *Mass Storage Systems and Technologies (MSST), 2012 IEEE 28th Symposium on*, pages 1–11. IEEE, 2012.

[82] Qing Liu, Jeremy Logan, Yuan Tian, Hasan Abbasi, Norbert Podhorszki, Jong Youl Choi, Scott Klasky, Roselyne Tchoua, Jay Lofstead, Ron Oldfield, Manish Parashar, Nagiza Samatova, Karsten Schwan, Arie Shoshani, Matthew Wolf, Kesheng Wu, and Weikuan Yu. Hello adios: the challenges and lessons of developing leadership class i/o frameworks. *Concurrency and Computation: Practice and Experience*, 26(7):1453–1473, 2014.

[83] Li-ta Lo, Christopher Sewell, and James P Ahrens. Piston: A portable cross-platform framework for data-parallel visualization operators. In *EGPGV*, pages 11–20, 2012.

[84] Benjamin Lorendeau, Yvan Fournier, and Alejandro Ribes. In-situ visualization in fluid mechanics using catalyst: A case study for code saturne. In *LDAV*, pages 53–57, 2013.

[85] Burlen Loring, Andrew Myers, David Camp, and E Wes Bethel. Python-based in situ analysis and visualization. In *Proceedings of the Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization*, pages 19–24, 2018.

[86] F. Løvholt, S. Lorito, J. Macias, M. Volpe, J. Selva, and S. Gibbons. Urgent tsunami computing. In *2019 IEEE/ACM HPC for Urgent Decision Making (UrgentHPC)*, pages 45–50, Nov 2019.

[87] Kwan-Liu Ma. In situ visualization at extreme scale: Challenges and opportunities. *IEEE Computer Graphics and Applications*, 29(6):14–19, 2009.

[88] Kwan-Liu Ma, Chaoli Wang, Hongfeng Yu, and Anna Tikhonova. In-situ processing and visualization for ultrascale simulations. In *Journal of Physics: Conference Series*, volume 78, page 012043. IOP Publishing, 2007.

[89] Preeti Malakar et al. Optimal execution of co-analysis for large-scale molecular dynamics simulations. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 60. IEEE Press, 2016.

[90] Preeti Malakar, Venkatram Vishwanath, Todd Munson, Christopher Knight, Mark Hereld, Sven Leyffer, and Michael E Papka. Optimal scheduling of in-situ analysis for large-scale scientific simulations. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 52. ACM, 2015.

[91] AC Mallinson, David A Beckingsale, WP Gaudin, JA Herdman, JM Levesque, and Stephen A Jarvis. Cloverleaf: Preparing hydrodynamics codes for exascale. *The Cray User Group*, 2013, 2013.

[92] J. Mandel, M. Vejmelka, A. Kochanski, A. Farguell, J. Haley, D. Mallia, and K. Hilburn. An interactive data-driven hpc system for forecasting weather, wildland fire, and smoke. In *2019 IEEE/ACM HPC for Urgent Decision Making (UrgentHPC)*, pages 35–44, Nov 2019.

[93] Jeremy S Meredith, Sean Ahern, Dave Pugmire, and Robert Sisneros. EAVL: the extreme-scale analysis and visualization library. In *Eurographics Symposium on Parallel Graphics and Visualization*, pages 21–30. The Eurographics Association, 2012.

[94] Jeremy S. Meredith, Robert Sisneros, David Pugmire, and Sean Ahern. A distributed data-parallel framework for analysis and visualization algorithm development. In *Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units*, GPGPU-5, pages 11–19, New York, NY, USA, 2012. ACM.

[95] Christopher Mitchell, James Ahrens, and Jun Wang. Visio: Enabling interactive visualization of ultra-scale, time series data via high-bandwidth distributed i/o systems. In *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 68–79. IEEE, 2011.

[96] Steven Molnar, Michael Cox, David Ellsworth, and Henry Fuchs. A sorting classification of parallel rendering. *Computer Graphics and Applications, IEEE*, 14(4):23–32, 1994.

[97] K. Moreland, U. Ayachit, B. Geveci, and Kwan-Liu Ma. Dax toolkit: A proposed framework for data analysis and visualization at extreme scale. In *Large Data Analysis and Visualization (LDAV), 2011 IEEE Symposium on*, pages 97–104, Oct 2011.

[98] Kenneth Moreland. Icet users' guide and reference. Technical Report 2011-5011, Sandia National Laboratory, 2011.

[99] Kenneth Moreland. Oh, $#*@! exascale! the effect of emerging architectures on scientific discovery. In *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*, pages 224–231. IEEE, 2012.

[100] Kenneth Moreland, Berk Geveci, Kwan-Liu Ma, and Robert Maynard. A classification of scientific visualization algorithms for massive threading. In *Proceedings of the 8th International Workshop on Ultrascale Visualization*, page 2. ACM, 2013.

[101] Kenneth Moreland, Wesley Kendall, Tom Peterka, and Jian Huang. An image compositing solution at scale. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, page 25. ACM, 2011.

[102] Kenneth Moreland, Ron Oldfield, Pat Marion, Sebastien Jourdain, Norbert Podhorszki, Venkatram Vishwanath, Nathan Fabian, Ciprian Docan, Manish Parashar, Mark Hereld, and M Papka. Examples of in transit visualization. In *Proceedings of the 2nd international workshop on Petascale data analytics: challenges and opportunities*, pages 1–6. ACM, 2011.

[103] Kenneth Moreland, Christopher Sewell, William Usher, Lita Lo, Jeremy Meredith, David Pugmire, James Kress, Hendrik Schroots, Kwan-Liu Ma, Hank Childs, Matthew Larsen, Chun-Ming Chen, Robert Maynard, and Berk Geveci. VTK-m: Accelerating the Visualization Toolkit for Massively Threaded Architectures. *IEEE Computer Graphics and Applications (CG&A)*, 36(3):48–58, May/June 2016.

[104] Kenneth Moreland, Brian Wylie, and Constantine 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*, pages 85–92. IEEE Press, 2001.

[105] Dmitriy Morozov and Zarija Lukic. Master of puppets: cooperative multitasking for in situ processing. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, pages 285–288. ACM, 2016.

[106] Carl Mueller. The sort-first rendering architecture for high-performance graphics. In *Proceedings of the 1995 symposium on Interactive 3D graphics*, pages 75–ff. ACM, 1995.

[107] Jurriaan D Mulder, Jarke J van Wijk, and Robert van Liere. A survey of computational steering environments. *Future generation computer systems*, 15(1):119–129, 1999.

[108] JR Neely. The us government role in hpc: Technology, mission, and policy. Technical report, Lawrence Livermore National Laboratory (LLNL), Livermore, CA, 2014.

[109] Ulrich Neumann. Communication costs for parallel volume-rendering algorithms. *Computer Graphics and Applications, IEEE*, 14(4):49–58, 1994.

[110] Arifa Nisar, Wei-keng Liao, and Alok Choudhary. Scaling parallel i/o performance through i/o delegate and caching system. In *2008 SC-International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12. IEEE, 2008.

[111] Ron A Oldfield, Kenneth Moreland, Nathan Fabian, and David Rogers. Evaluation of methods to integrate analysis into a large-scale shock shock physics code. In *Proceedings of the 28th ACM international conference on Supercomputing*, pages 83–92. ACM, 2014.

[112] Steven G Parker and Christopher R Johnson. Scirun: a scientific programming environment for computational steering. In *Proceedings of the 1995 ACM/IEEE conference on Supercomputing*, page 52. ACM, 1995.

[113] Valerio Pascucci, Daniel E Laney, Ray J Frank, Giorgio Scorzelli, Lars Linsen, Bernd Hamann, and Francois Gygi. Real-time monitoring of large scientific simulations. In *Proceedings of the 2003 ACM symposium on Applied computing*, pages 194–198. ACM, 2003.

[114] Tom Peterka, David Goodell, Robert Ross, Han-Wei Shen, and Rajeev Thakur. A configurable algorithm for parallel image-compositing applications. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, page 4. ACM, 2009.

[115] David Pugmire, James Kress, Hank Childs, Matthew Wolf, Greg Eisenhauer, Randy Churchill, Tahsin Kurc, Jong Choi, Scott Klasky, Kesheng Wu, Alex Sim, and Junmin Gu. Visualization and analysis for near-real-time decision making in distributed workflows. In *High Performance Data Analysis and Visualization (HPDAV) 2016 held in conjuction with IPDPS 2016*, May 2016.

[116] David Pugmire, James Kress, Jeremy Meredith, Norbert Podhorszki, Jong Choi, and Scott Klasky. Towards scalable visualization plugins for data staging workflows. In *Big Data Analytics: Challenges and Opportunities (BDAC-14) Workshop at Supercomputing Conference*, November 2014.

[117] Daniel A Reed and Jack Dongarra. Exascale computing and big data. *Communications of the ACM*, 58(7):56–68, 2015.

[118] Marzia Rivi, Luigi Calori, Giuseppa Muscianisi, and Vladimir Slavnic. In-situ visualization: State-of-the-art and some use cases. *PRACE White Paper; PRACE: Brussels, Belgium*, 2012.

[119] Robert Rosner and Ernie Moniz. Science based nuclear energy systems enabled by advanced modeling and simulation at the extreme scale. In *ASCR Scientific Grand Challenges Workshop Series, Tech. Rep*, 2009.

[120] Kiminori Sato, Kathryn Mohror, Adam Moody, Todd Gamblin, Bronis R de Supinski, Naoya Maruyama, and Shingo Matsuoka. A user-level infiniband-based file system and checkpoint strategy for burst buffers. In *Cluster, Cloud and Grid Computing (CCGrid), 2014 14th IEEE/ACM International Symposium on*, pages 21–30. IEEE, 2014.

[121] Will J Schroeder, Bill Lorensen, and Ken Martin. *The visualization toolkit*. Kitware, 2004.

[122] William J Schroeder, Kenneth M Martin, and William E Lorensen. The design and implementation of an object-oriented toolkit for 3d graphics and visualization. In *Proceedings of the 7th conference on Visualization'96*, pages 93–ff. IEEE Computer Society Press, 1996.

[123] Christopher Sewell, Jeremy Meredith, Kenneth Moreland, Tom Peterka, Dave DeMarle, Li-ta Lo, James Ahrens, Robert Maynard, and Berk Geveci. The sdav software frameworks for visualization and analysis on next-generation multi-core and many-core architectures. In *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*, pages 206–214. IEEE, 2012.

[124] G. Shipman, S. Campbell, D. Dillow, M. Doucet, J. Kohl, G. Granroth, R. Miller, D. Stansberry, T. Proffen, and R. Taylor. Accelerating data acquisition, reduction, and analysis at the spallation neutron source. In *2014 IEEE 10th International Conference on e-Science*, volume 1, pages 223–230, Oct 2014.

[125] Magdalena Slawinska, Michael Clark, Matthew Wolf, Tanja Bode, Hongbo Zou, Pablo Laguna, Jeremy Logan, Matthew Kinsey, and Scott Klasky. A maya use case: adaptable scientific workflows with adios for general relativistic astrophysics. In *Proceedings of the Conference on Extreme Science and Engineering Discovery Environment: Gateway to Discovery*, page 54. ACM, 2013.

[126] Rick Stevens and Mark Ellisman. Opportunities in biology at the extreme scale of computing. In *ASCR Scientific Grand Challenges Workshop Series, Tech. Rep*, 2009.

[127] Akira Takeuchi, Fumihiko Ino, and Kenichi Hagihara. An improved binary-swap compositing for sort-last parallel rendering on distributed memory multiprocessors. *Parallel Computing*, 29(11):1745–1762, 2003.

[128] William Tang, David Keyes, N Sauthoff, N Gorelenkov, J Cary, A Kritz, S Zinkle, J Brooks, R Betti, and W Mori. Scientific grand challenges: Fusion energy sciences and the role of computing at the extreme scale. In *DoE-SC Peer-reviewed report on major workshop held March*, pages 18–20, 2009.

[129] Roselyne Tchoua, Jong Choi, Scott Klasky, Qing Liu, Jeremy Logan, Kenneth Moreland, Jingqing Mu, Manish Parashar, Norbert Podhorszki, David Pugmire, and Matthew Wolf. Adios visualization schema: A first step towards improving interdisciplinary collaboration in high performance computing. In *eScience (eScience), 2013 IEEE 9th International Conference on*, pages 27–34. IEEE, 2013.

[130] The HDF Group. Hdf5 users guide. `https://www.hdfgroup.org/HDF5/doc/UG/`. Accessed: 6/20/2016.

[131] David Thompson, Jeff Braun, and Ray Ford. *OpenDX: paths to visualization; materials used for learning OpenDX the open source derivative of IBM's visualization Data Explorer*. Visualization and Imagery Solutions, 2004.

[132] Anna Tikhonova, Carlos D Correa, and Kwan-Liu Ma. Visualization by proxy: A novel framework for deferred interaction with volume data. *IEEE Transactions on Visualization and Computer Graphics*, 16(6):1551–1559, 2010.

[133] Anna Tikhonova, Hongfeng Yu, Carlos D Correa, Jacqueline H Chen, and Kwan-Liu Ma. A preview and exploratory technique for large-scale scientific simulations. In *EGPGV*, pages 111–120, 2011.

[134] Craig Upson, TA Faulhaber, David Kamins, David Laidlaw, David Schlegel, Jefrey Vroom, Robert Gurwitz, and Andries Van Dam. The application visualization system: A computational environment for scientific visualization. *IEEE Computer Graphics and Applications*, 9(4):30–42, 1989.

[135] Jeffrey Vetter and Karsten Schwan. High performance computational steering of physical simulations. In *Parallel Processing Symposium, 1997. Proceedings., 11th International*, pages 128–132. IEEE, 1997.

[136] Venkatram Vishwanath, Mark Hereld, Vitali Morozov, and Michael E Papka. Topology-aware data movement and staging for i/o acceleration on blue gene/p supercomputing systems. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, page 19. ACM, 2011.

[137] Venkatram Vishwanath, Mark Hereld, Michael E Papka, Randy Hudson, G Cal Jordan IV, and C Daley. In situ data analysis and i/o acceleration of flash astrophysics simulation on leadership-class system using glean. In *Proc. SciDAC, Journal of Physics: Conference Series*, 2011.

[138] W Washington. Challenges in climate change science and the role of computing at the extreme scale. In *Proc. of the Workshop on Climate Science*, 2008.

[139] Brad Whitlock, Jean M Favre, and Jeremy 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*, pages 101–109. Eurographics Association, 2011.

[140] Brad J Whitlock, Steve M Legensky, and Jim Forsythe. In situ infrastructure enhancements for data extract generation. In *54th AIAA Aerospace Sciences Meeting*, page 1928, 2016.

[141] Jonathan Woodring, J Ahrens, J Figg, Joanne Wendelberger, Salman Habib, and Katrin Heitmann. In-situ sampling of a large-scale particle simulation for interactive visualization and analysis. In *Computer Graphics Forum*, volume 30, pages 1151–1160. Wiley Online Library, 2011.

[142] Don-Lin Yang, Jen-Chih Yu, and Yeh-Ching Chung. Efficient compositing methods for the sort-last-sparse parallel volume rendering system on distributed memory multicomputers. *The Journal of Supercomputing*, 18(2):201–220, 2001.

[143] Yucong Ye, Robert Miller, and Kwan-Liu Ma. In situ pathtube visualization with explorable images. In *Proceedings of the 13th Eurographics Symposium on Parallel Graphics and Visualization*, pages 9–16. Eurographics Association, 2013.

[144] Glenn Young, David Dean, and Martin Savage. Forefront questions in nuclear science and the role of high performance computing. In *Technical report, ASCR Scientific Grand Challenges Workshop Series*, 2009.

[145] Hongfeng Yu, Chaoli Wang, Ray W Grout, Jacqueline H Chen, and Kwan-Liu Ma. In situ visualization for large-scale combustion simulations. *IEEE Computer Graphics and Applications*, pages 45–57, 2010.

[146] Hongfeng Yu, Chaoli Wang, and Kwan-Liu Ma. Massively parallel volume rendering using 2–3 swap image compositing. In *High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for*, pages 1–11. IEEE, 2008.

[147] Fan Zhang, Tong Jin, Qian Sun, Melissa Romanus, Hoang Bui, Scott Klasky, and Manish Parashar. In-memory staging and data-centric task placement for coupled scientific simulation workflows. *Concurrency and Computation: Practice and Experience*, 29(12), 2017.

[148] Fang Zheng, Hasan Abbasi, Ciprian Docan, Jay Lofstead, Qing Liu, Scott Klasky, Manish Parashar, Norbert Podhorszki, Karsten Schwan, and Matthew Wolf. Predata–preparatory data analytics on peta-scale machines. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–12. IEEE, 2010.

[149] Fang Zheng, Hongbo Zou, J Cao, J Dayal, T Nugye, G Eisenhauer, and S Klasky. Flexio: location-flexible execution of in-situ data analytics for large scale scientific applications. In *Proceedings IEEE international parallel and distributed processing symposium (IPDPS'13)*, pages 320–331, 2013.

[150] Hongbo Zou, Yongen Yu, Wei Tang, and Hsuanwei Michelle Chen. Improving i/o performance with adaptive data compression for big data applications. In *Parallel & Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International*, pages 1228–1237. IEEE, 2014.