MIDDLEWARE FOR ONLINE SCIENTIFIC DATA ANALYTICS AT EXTREME SCALE

A Thesis
Presented to
The Academic Faculty

by

Fang Zheng

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
College of Computing, School of Computer Science

Georgia Institute of Technology
May 2014

MIDDLEWARE FOR ONLINE SCIENTIFIC DATA ANALYTICS AT EXTREME SCALE

Approved by:

Professor Karsten Schwan,
Committee Chair
College of Computing, School of Computer Science
Georgia Institute of Technology

Professor Karsten Schwan, Advisor
College of Computing, School of Computer Science
Georgia Institute of Technology

Dr. Scott Klasky
Computer Science and Mathematics Division
Oak Ridge National Laboratory

Professor Ling Liu
College of Computing, School of Computer Science
Georgia Institute of Technology

Professor Richard Vuduc
College of Computing, School of Computational Science and Engineering
Georgia Institute of Technology

Dr. Matthew Wolf
College of Computing, School of Computer Science
Georgia Institute of Technology

Date Approved: 5 March 2014

Dedicated to my wife and parents.

ACKNOWLEDGEMENTS

First of all, I want to thank my advisor, Prof. Karsten Schwan, for his guidance and support throughput my PhD studies. Karsten introduced me to the research field, and gives me the freedom to explore various topics. His vision and insights have proven to be invaluable. Besides, Karsten is very supportive and patient, which means a lot when things did not work out. I will always cherish the opportunity to work with him.

I am also very grateful to Dr. Matthew Wolf. Matt has put significant amounts of time in my work, and his guidance always pushes me to think deeper and sharper. Matt is very accessible and gives me a lot of hands-on help.

I would like to express my deep gratitude to Prof. Ling Liu and Prof. Richard Vuduc for serving on my thesis committee. Their insightful feedback has significantly improved this thesis.

Dr. Scott Klasky is my mentor during my first and second summer internships in Oak Ridge National Laboratory, and has been a great collaborator since then. Scott is one of the most hard-working people I have ever met. He always encourages me to pursue problems which matter in reality. He also offers me access to the computational resources and real-world use cases without which this thesis would be impossible.

Dr. Greg Eisenhauer helps me a lot on programming and implementation. I am always amazed by Greg's programming skills and deep understanding of the systems. Besides, Greg is very patient and responsive whenever I need his help.

Dr. Chitra Venkatramani, my mentor at IBM Research, gives me tremendous support and help in my research and future career. Chitra is always encouraging, and believes I can do good work.

During my PhD studies, I have the privilege to work with many brilliant people. Jay Lofstead and I worked together for three years, and he has been a great mentor and friend to me. Jay is always there when I need help. Dr. Hasan Abbasi gave me tremendous help and our collaboration has been productive along the way. Qing Liu, Dr. Nobert Podhorszki, and Roselyne Barreto at Oak Ridge National Laboratory made my interships at ORNL very enjoyable experieinces. I also benefited and learned a lot from my collaborators at IBM Research, including my mentor Sujy Parekh, Rohit Wagle, Yoonho Park, Richard King, Philippe Selo, Kun-Lung Wu, and Bugra Gedik.

I thank my fellow students in CERCS research group, Chengwei Wang, Hongbo Zou, Hrishikesh Amur, Jianting Cao, Jai Dayal, Liting Hu, Sudarsun Kannan, Mukil Kesavan, Mahendra Kutare, Minsung Jang, Min Lee, Alexander Merritt, Adit Ranadive, Priyanka Tembey, and Hobin Yoon. They help me in many ways and make my stay at Atlanta a lot more fun. I am also grateful to our Administrator Coordinator, Mrs. Susie McClain, for her help in the past seven years.

I have made great friends in Atlanta. I will always cherish my friendship with Chengwei Wang, Hongbo Zou, Qingyang Wang, Yuzhe Tang, Xuechen Zhang, Qinyi Wu, Yong Zhao, Jianyong Xie, Fei Sun, and Yinghui Zhang.

Finally, I would like to thank my wife and parents for their love and support. My wife, Lu Tang, is my source of inspiration. She is always there to cheer me up. My parents, Zhenghui Zheng and Guohua Fu, give me the role model to follow. Their love and guidance shapes me into the person I am. This theis is dedicated to them.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Scientific simulations running on High End Computing machines in domains like Fusion, Astrophysics, and Combustion now routinely generate terabytes of data in a single run, and these data volumes are only expected to increase. Since such massive simulation outputs are key to scientific discovery, the ability to rapidly store, move, analyze, and visualize data is critical to scientists' productivity. Yet there are already serious I/O bottlenecks on current supercomputers, and movement toward the Exascale is further accelerating this trend.

This dissertation is concerned with the design, implementation, and evaluation of middleware-level solutions to enable high performance and resource efficient online data analytics to process massive simulation output data at large scales. Online data analytics can effectively overcome the I/O bottleneck for scientific applications at large scales by processing data as it moves through the I/O path. Online analytics can extract valuable insights from live simulation output in a timely manner, better prepare data for subsequent deep analysis and visualization, and gain improved performance and reduced data movement cost (both in time and in power) compared to the conventional post-processing paradigm.

The thesis identifies the key challenges for online data analytics based on the needs of a variety of large-scale scientific applications, and proposes a set of novel and effective approaches to efficiently program, distribute, and schedule online data analytics along the critical I/O path. In particular, its solution approach i) provides a high performance data movement substrate to support parallel and complex data exchanges between simulation and online data analytics, ii) enables placement flexibility of analytics to exploit distributed resources, iii) for co-placement of analytics

with simulation codes on the same nodes, it uses fined-grained scheduling to harvest idle resources for running online analytics with minimal interference to the simulation, and finally, iv) it supports scalable efficient online spatial indices to accelerate data analytics and visualization on the deep memory hierarchies of high end machines.

Our middleware approach is evaluated with leadership scientific applications in domains like Fusion, Combustion, and Molecular Dynamics, and on different High End Computing platforms. Substantial improvements are demonstrated in end-to-end application performance and in resource efficiency at scales of up to 16384 cores, for a broad range of analytics and visualization codes. The outcome is a useful and effective software platform for online scientific data analytics facilitating large-scale scientific data exploration.

CHAPTER I

INTRODUCTION

1.1 Problem Statement

1.1.1 Need for Online Scientific Data Analytics

The last decade has witnessed explosive increases in data in many science domains. Thanks to the tremendous technical advances in computation and instruments, scientists can now obtain data at unprecedented scales and fidelity. For example, scientific simulations running on High End Computing machines in domains like Fusion, Astrophysics, and Combustion now routinely generate terabytes of data in a single run, and these data volumes are only expected to increase. The data carries information of the subjects being investigated, and is essential for scientific discoveries. Consequently, the ability to rapidly store, move, analyze, and visualize data is critical to scientists' productivity.

The "Big Data" generated by scientific simulations and instruments imposes steadily increasing pressure on the I/O and storage sub-systems associated with such facilities. In fact, for high end simulations, I/O is now widely recognized as a severe performance bottleneck for both the simulation and data post-processing. This is because while simulations generate ever larger data volumes to be analyzed/visualized, there is an increasingly large disparity between the I/O and computational capabilities on most High-End Computing (HEC) machines [108]. Factors like contention on shared resources [90] and complicated I/O patterns [89] further exacerbate the attainable I/O performance on those platforms. The combined effect results in undesirable situations where a substantial portion of the simulation runtime is spent in writing data to the storage system [108]. Furthermore, scientifically important analysis and

visualization on the output data incur considerable I/O overhead (e.g., it has been reported that data read times from storage can consume up to 98% of the total run-time for large-scale visualizations [34]). Another important issue is the undue power consumption of large and/or repeated data movements into and out of storage [127]. Unfortunately, such I/O bottlenecks are only expected to exaggerate

Online data analytics has emerged as an effective way to overcome the I/O bottleneck for scientific applications running at the Peta-Scale and beyond. In this paradigm, data analytics is deployed on the same HEC platform where the simulation runs, with simulation output data processed while it is being generated. Compared to conventional post-processing methods that first write data to storage and then read it back for analysis, online data analytics has the following significant advantages:

First, online data analytics can significantly reduce on-machine data movement and disk I/O activities, which results in reduced simulation and data processing runtimes.

Second, online analytics can extract valuable insights from live simulation output in a more timely manner, which makes it particularly useful for real-time simulation monitoring, data diagnostics, and computational steering.

Third, by processing data as it moves through the I/O path, online data analytics can better prepare data for subsequent deep analysis and visualization.

Substantial previous work in our team and elsewhere has demonstrated that many useful analytics can be performed in an online fashion. Typical use cases include data reduction, compression and filtering prior to data movement, data re-organization for improved storage access or to enable useful analysis, and data processing and analysis to monitor simulation progress or status and to provide end users with rapid insight into scientific outcomes produced by simulations. In fact, the utility of online data analytics is evident from its wide adoption of leading scientific applications, including the S3D combustion simulation [1, 162], the GTC [70, 160] and GTS [139, 162] fusion

simulations, CHIMERA astrophysics simulation [11], Trillions [106], CTH [71, 67], and FLASH [50, 94]. Given the increasingly high volumes of output data generated by scientific simulations, we expect that online data analytics will become common practice for applications running on current peta-scale and next generation high end machines and we note that this trend in high performance computing is mirrored by similar recent work in efficient "stream data processing" in the enterprise domain [19, 141, 142, 143].

1.1.2 The State of The Art

Online data analytics and visualization have gained much recent attention from the Scientific Computing community. Current work falls into two categories: 1) novel online data analytics and visualization algorithms, including indexing [35, 69], compression [73], feature extraction [3], and various visualization techniques [151, 102]; and 2) supporting tools and infrastructures. The first category describes the target workloads we aim to support. Regarding the second category, a number of systems and tools have been developed to support online data analytics and visualization, and they can be further categorized according to their placement strategies of online analytics along the I/O path. As shown in Figure 1, simulation output data originates in compute nodes, is moved via the interconnect to the I/O servers of the parallel file system, and finally reaches back-end persistent storage. Existing supporting infrastructures have explored various options to place online analytics along the I/O path.

1) Analytics and Visualization Libraries. ParaView's co-processing library [102], VisIt's remote visualization [20], and other online visualization work [151, 138] offer software libraries with collections of analytics and visualization routines. Although those libraries do not impose restrictions on where they can be executed and can be directly invoked as function calls, the libraries must rely on external data movement

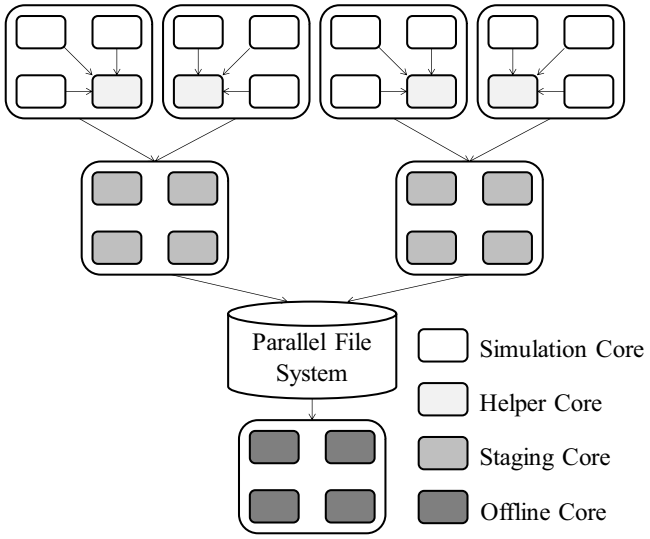


Figure 1: Online data analytics along I/O path.

support to run in staging nodes or on dedicated compute node cores, termed 'helper' cores in Figure 1.

2) Helper Core Processing. Functional partitioning [81], software accelerator [129], and Damaris [43] perform file I/O and analytics on dedicated cores in compute nodes and leverage shared memory to pass data from simulation to analytics. Such approach can be beneficial for some cases (e.g., when the simulation cannot scale to use all cores), but its applicability is restricted by the memory space on compute nodes made available by the simulation.

3) Staging Area Processing. Data Services [11], Nessie [61], GLEAN [94], and HDF5/DSM [23] use a set of additional staging nodes to buffer and process simulation output data. Placing analytics on staging nodes requires provisioning additional nodes, and moving massive simulation output data to staging area can be costly and negatively interfere with simulation [12].

4) Active Storage. Certain analytics routines may be deployed directly on I/O servers and triggered to operate when data is written and/or read [112, 131]. Due to resource limitations on I/O servers, the deployed analytics are usually restricted kernel

functions. Further, access to storage nodes is not generally allowed in production environments.

5) Offline Processing. Data written to storage is read back for additional or long-term analysis or visualization [51], typically assisted by workflow tools [93, 40].

6) Hybrid Online Processing. As with our work, DataSpaces [1] permits analytics to be broken down into separate pieces and deployed onto both compute nodes and staging nodes.

1.1.3 Challenges

Despite increased adoption by real-world applications, there exist significant challenges to enable online data analytics at large scales and achieve satisfactory performance.

[**High Performance Data Movement Substrate**] Online data analytics must consume live simulation output data at runtime. Since the simulations typically run at large scale and can frequently generate massive volumes of data, data movement between simulation and analytics can significantly impact overall performance. In most applications, simulation and analytics are implemented and run as separate parallel programs for performance and/or software engineering purposes. In this case, large data volumes are moved in many concurrent streams between sets of communicating processes. Such data exchanges can be complex, in part due to the potential mismatches of the data distributions and formats used by the simulation vs. by analytics, resulting in complex and diverse data exchange patterns. To offload from scientists the complex and error-prone task to implement and tune such complex data exchanges, one of the major requirements for enabling online data analytics is a data movement substrate that can support complex data exchange patterns with high performance and scalability, by providing higher level abstractions to describe these data exchanges. Implementing such a data movement facility is non-trivial. In

fact, existing parallel code coupling tools are shown to have efficiency issues at large scales [155], and only a few online analytics supporting platforms show scalability beyond a couple of thousand cores or provide easy-to-use programming interfaces to facilitate coupling simulations with their analytics.

[**Analytics Placement Flexibility**] For online processing of the outputs generated by large scale simulations, a key problem to address is "where" analytics are placed along the I/O path (as illustrated in Figure 1): on compute nodes integrated with application codes, on compute nodes as separate software components, on nodes dedicated to analytics (also termed 'staging nodes'), or offline (after data is placed into persistent storage). Placing data analytics involves deciding the resources to allocate to analytics computation and realizing the data movements between simulation and analytics. Our experimental results and analytical models [159] show that analytics placement can significantly impact the performance (e.g., runtime) and cost (e.g., CPU hours) of the coupled simulation and analytics and that the best placement depends on the particular analytics codes, data volumes, scale of operation, and machine characteristics. The consequent insight is that no single, specific placement will be the "best" for all applications and analytics.

Such variation makes flexible placement a critical element of any infrastructure supporting online analytics. On the one hand, scientists desire the performance benefit from good placements, but it is a burden for them to tune placements every time a different analytics code is run, especially when this requires significant coding effort. There is a need, therefore, for infrastructure that makes it easy to decide, enforce, change, and tune analytics placement. On the other hand, if such an analytics software infrastructure aims to support a broad range of simulations and analytics, lacking placement flexibility limits its applicability, since fixed placements may negatively impact application performance at large scale. Unfortunately, most existing online analytics systems only support certain, fixed placement choices and therefore,

each is efficient for or applicable to only certain classes of analytics. Some permit analytics to run at different locations, but require adopting particular coding patterns or re-placement involves substantial re-coding.

[**End-to-End Performance and Cost Tuning**] When jointly running a simulation with its associated online data analytics, there is a producer-consumer data dependency between the simulation and analytics components, effectively causing them to form a so-called "I/O pipeline". This pipeline's overall performance is not only determined by its slowest component, but it is also impacted by the interference between concurrently running components. Such a coupled execution model calls for tuning techniques that take into account the end-to-end performance and cost of the entire I/O pipeline, in addition to those of its individual elements. In particular, the execution rates of the simulation and of analytics should be balanced to reduce pipeline stalls; otherwise, the bottlenecking component would not only inhibit overall pipeline performance, but would also cause resources to be wasted in other components (e.g., the large-scale simulation). Another issue is the need to manage the potential interference between simulation and analytics, to reduce negative performance loss. Such interference can be due to contention on shared resources, including the shared interconnect and on-node memory resources (e.g., shared last level cache and memory bus bandwidth). As we will show in later chapters, existing work on online data analytics often fails to consider the coupled simulation and analytics as a whole, and is therefore, unable to deal with such issues.

[**Coping with Architectural Trends**] In order to achieve desired high performance and scalability, the online analytics system should effectively exploit hardware architecture features, leveraging ongoing trends in architecture evolution. In particular, three notable architectural features are relevant to online data analytics.

The first concerns high performance interconnects like InfiniBand and Cray's Gemini. Those interconnects commonly provide RDMA capability for low-latency, high

throughput data movement, and should be leveraged for moving large amounts of data between simulation and analytics.

The second is the Chip Multi-Processor (CMP, or multi-core) architecture that is pervasive on today's HPC machines and is expected to remain so for the foreseeable future. CMP architectures feature not only many CPU cores for parallel execution, but also diverse memory resources (cache, memory controller, memory bus, etc.) shared among concurrently executing threads. The implications for online data analytics include: i) the parallelism provided by multi-cores makes it possible to map simulation and analytics to different cores and running them concurrently to exploit pipeline parallelism; and ii) the contention on shared memory resources among co-located simulation and analytics should be mitigated to reduce negative interference.

The third is the deep memory hierarchy equipped with Non-Volatile Memory such as Flash SSD (Solid State Disks). DRAM is becoming increasingly expensive (in terms of power consumption) and scarce (in terms of per-core DRAM capacity). SSD, on the other hand, has advantages like high I/O performance and low power consumption. It has been shown to be both feasible and beneficial to incorporate SSDs into the node-local memory hierarchy to improve the node's performance/cost point. However, there has not yet been much work on porting and optimizing online data analytics to SSD-equipped memory hierarchies, and the potential and limitations of doing remain open questions.

[**Scalable Online Spatial Indices** Many analytics and visualization tasks can be accelerated by spatial indices. A spatial index partitions a multi-dimensional coordinates or attribute space into a bounded hierarchy, and can help quickly locate regions of interest in massive data sets. Popular spatial indices like RTree, Octree, and KD-Tree are commonly used in many types of analytics, including volume rendering, iso-surface, N-point correlation, kernel summation, feature tracking, and so forth.

To use spatial indices for online analytics, however, there are several challenges to address. First, the index must be constructed from live simulation output data of high volume and velocity. This calls for index construction methods that can operate in a streaming manner, where the overheads of generating, buffering and transferring spatial indices need to be carefully managed and tuned to avoid overwhelming the overall I/O pipeline. Further, since most analytics run as parallel programs and their performance is sensitive to data distribution, they can suffer severely from data skews. This implies that the spatial indices (along with the original data) must be properly distributed and should achieve load balance for queries, even with skewed data distributions. Although there exists extensive prior work on distributed spatial indices in the context of databases and geographical information systems, little work has been done on constructing, distributing, and querying online spatial indices for scientific simulation output data, to accommodate its streaming nature, its high data volumes, and its massive scale. Furthermore, the emerging deep memory hierarchy requires index structures to be space efficient and index construction and queries on these structures to efficiently manage data movements between the different levels of the memory hierarchy present on high end machines.

1.2 Thesis Statement

Motivated by the importance and challenges of online scientific data analytics, this dissertation addresses the following thesis statement:

A middleware that provides high performance data movement, flexible analytics placement, interference management, and support for online spatial indices can enable high performance and resource efficient online data analytics to process massive simulation output data at large scales.

In order to demonstrate this statement, the thesis identifies the technical challenges in implementing, optimizing, and scaling online scientific data analytics, develops middleware-level solutions to address those challenges, applies the solutions to multiple representative real-world applications, and experimentally measures the benefits at large scales on leadership High End Computing computers.

1.3 Technical Contributions

In this dissertation, we make the following technical contributions:

1) Targeting a common class of online data analytics termed as Preparatory Data Analytics, we propose the PreDatA middleware for preparing and characterizing data while it is being produced by the large scale simulations running on peta-scale machines. By dedicating additional compute nodes on the machine as "staging" nodes and by staging simulations' output data through these nodes, PreDatA can exploit their computational power to perform online analytics with lower latency than attainable by first moving data into file systems and storage. PreDatA uses RDMA-based asynchronous data movement to reduce I/O latency. It offers a MapReduce-like programming model for application-specific operations on streaming data that can discover latent data characteristics and/or appropriately reorganize and annotate data to speed up subsequent post-processing. PreDatA is useful for data pre-processing, runtime data analysis and inspection, as well as for data exchanges between concurrently running simulations. Performance evaluations with several production peta-scale applications on up to 16384 cores demonstrate the applicability of the PreDatA approach to a broad set of useful analytics. They also show that use of PreDatA can lead to improved simulation time, timely insight into output data and improved read performance of output files.

2) We use both experiments and performance model to reveal that the placement of online data analytics can significantly impact end-to-end performance of analytics

pipelines and of the simulations 'feeding' them, with one resulting requirement being flexibility in the location of data analytics. Our response is the FlexIO I/O middleware that enables flexible analytics placement. FlexIO provides high performance, memory efficient intra- and inter-node data movement transports that permit diverse analytics placement options. It offers simple high-level programming interfaces that can be used to implement complex data exchange patterns and makes changes in analytics placement transparent to simulation and analytics codes. Various placement policies can be built on top of FlexIO to exploit location flexibility for tuning application performance, CPU usage, and data movement cost. FlexIO operates on both InfiniBand and the Cray XK6 with Gemini interconnect, and has been used to implement in situ analytics for two leadership scientific applications: the GTS fusion simulation and the S3D combustion simulation. Experiments show that leveraging the flexibility enabled by FlexIO to tune placement can improve total execution time by up to 30% compared to inline-only solutions and the benefit is more evident at larger scales.

3) To improve the resource efficiency of online data analytics, we propose a runtime approach that can run analytics using idle resources "stolen" from the simulation on compute nodes, without reducing simulation performance. We first characterize in detail the runtime behavior of six representative simulation codes. This reveals that there exist substantial unused idle resources in compute nodes during a typical simulation's execution. These findings lead to the creation of an agile runtime system called "GoldRush" that can harvest those otherwise-wasted, idle resources on compute nodes to efficiently perform data analytics co-located with the simulation. GoldRush uses fine-grained scheduling to "steal" idle resources from the simulation in ways that incur negligible runtime overheads and minimize interference between the simulation and analytics. This involves recognizing the potential causes of on-node resource contention and then using scheduling methods that prevent them. Experiments with

representative applications at large scales (up to 12288 cores on Hopper Cray XE6) show that resources harvested by GoldRush can be used to perform useful analytics, significantly improving resource efficiency, reducing data movement costs, and posing negligible impact on simulations.

4) We introduce ZStore, which provides a general and scalable framework to construct online spatial indices from live simulation output data, and offers the index structures needed by analytics for answering diverse spatial queries. ZStore uses a flexible in-transit index construction workflow embedded in the I/O path to leverage distributed resources for building the index in a streaming manner. The workflow is highly customizable to allow application-specific control in data distribution. ZStore also provides a SSD-optimized buffer management utility for building an out-of-core index on a deeper memory hierarchy. We implement two representative spatial indices: RTree and Octree, and for each index, we propose novel optimizations that significantly improve performance and memory efficiency.

As a body of work, this dissertation demonstrates a middleware solution to enable online analytics on massive scientific data. The solution achieves scalability, resource efficiency, and high end-to-end application performance.

From the science end users' perspective, this dissertation makes it possible to perform a wide range of analytics on live simulation output data in a timely and cost-efficient manner, enhances the processs of scientific discovery, and helps scientists cope with the data deluge on current and future High End Computing platforms.

1.4 Organization of the Dissertation

The remainder of this thesis dissertation is organized as follows.

Chapter II presents application requirements for online data analytics, based on use cases from several leadership scientific applications, discusses the limitations of previous work, and reasons about the design choices for our middleware solution.

Chapter III presents the PreDatA system that can be used for implementing preparatory data analytics, and demonstrates the scalability and performance benefits of PreDatA with two real-world applications on up to 16384 cores.

Chapter IV presents FlexIO, an I/O middleware providing high performance data movement between simulation and analytics, and enabling location flexibility for online analytics. We show that such placement flexibility can be exploited to effectively tune the end-to-end performance of typical analytics pipelines.

Chapter V presents the GoldRush runtime system, which can harvest resources unused by the simulation to run analytics on the same compute nodes, and can mitigate interference between both, thus significantly improving overall resource efficiency.

Chapter VI presents ZStore, a scalable and general framework for constructing and querying online spatial indices from live simulation output data. ZStore also supports efficient out-of-core indices on SSD-equipped deep memory hierarchies.

Chapter VII summarizes the dissertation, draws conclusions, and discusses open problems and future research directions.

CHAPTER II

BACKGROUND AND MOTIVATION

In this chapter, we first describe applications' essential requirements of online data analytics supporting platforms drawn from a variety of real-world workloads. We then develop an analytical performance model to capture the fundamental factors which can impact the end-to-end performance of coupled simulation and analytics. Based on the application requirements and performance model, we reason about the limitations of previous work and the design choices for our online data analytics middleware.

2.1 Application Requirements

Although online data analytics has been increasingly adopted by the Scientific Computing community, most applications have used rather ad-hoc and application-specific implementations and many important issues are not well understood yet. For instance, what is the common usage of online data analytics? What are the fundamental system-level factors impacting the performance of online data analytics? How far from the optimal performance are those existing solutions listed in Section 1.1.2? Can we support various online data analytics use cases in a way which is both general and superior to ad-hoc solutions in performance? To answer these questions, it is necessary to have a thorough understanding of application requirements for online data analytics supporting platforms.

In this section, we survey a set of use cases for online data analytics from real-world scientific applications. All these applications can scale to large core counts and require rapid analysis of large volumes of data. They are also diverse in their science domains, output data formats and volumes, as well as the specific analytics to perform. Most of the applications described here are popularly used and considered as

prominent peta-scale applications by DOE. Therefore, these applications constitute a representative set of use cases for online data analytics.

For each application, we focus on the following characteristics: i) application scale; ii) simulation I/O characteristics (output data format, volume and I/O frequency); and iii) the used analytics (purpose, computation model, scalability, data expansion or reduction ratio, etc.).

2.1.1 GTC

The first application is GTC. GTC (Gyrokinetic Toroidal Code) [70] is a 3-Dimensional Particle-In-Cell code used to study micro-turbulence in magnetic confinement fusion from first principles plasma theory. It outputs particle data that includes two 2D arrays for electrons and ions, respectively. Each row of the 2D array records eight attributes of a particle including coordinates, velocities, weight, and particle label. The last two attributes, process rank and particle local ID within the process, jointly form the label that globally identifies a particle. They are determined on each particle at the start of a simulation and remain unchanged throughout the particle's lifetime. These two arrays are distributed among all processes, and particles move across processes in a random manner as the simulation evolves, resulting in two out-of-order particle arrays. In a production run at the scale of 16,384 cores, each core can output two million particles roughly every 120 seconds resulting in 260GB of particle data per output.

As shown in Figure 2, three analysis and preparation tasks are performed on particle data. The first involves tracking across multiple iterations of a million-particle subset out of the billions of particles, requiring searching among the hundreds of 260GB output files by the particle label. To expedite this operation, particles can be (and for our example are) sorted by their labels before searching. The second task performs a range query to discover the particles whose coordinates fall into certain

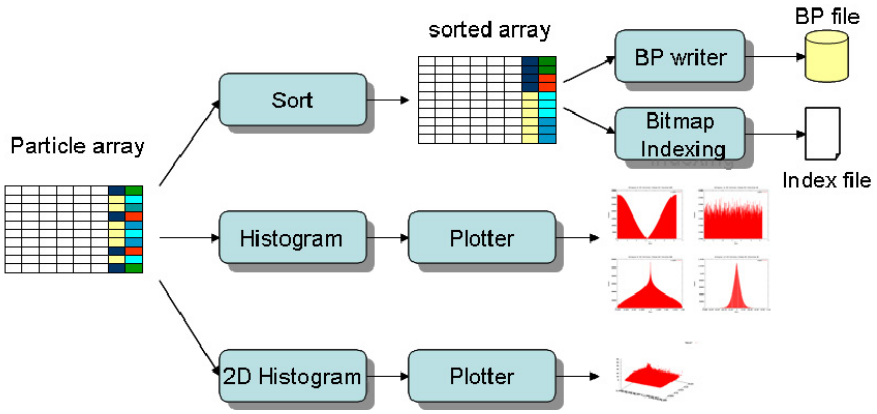


Figure 2: Illustration of online data analytics for GTC.

ranges. A bitmap indexing technique [130] is used to avoid scanning the whole particle array, and multiple array chunks are merged to speed up bulk loading. The third task is to generate 1D histograms and 2D histograms on attributes of particles [64] to enable online monitoring of the running GTC simulation. 2D histograms can also be used for visualizing parallel coordinates [64] in subsequent data analysis.

2.1.2 GTS

The second application, GTS (Gyrokinetic Tokamak Simulation), is a global three-dimensional Particle-In-Cell (PIC) code used to study the microturbulence and associated transport in magnetically confined fusion plasma of tokamak torodial devices [139]. Similar to GTC, GTS simulation outputs particle data containing two 2-dimensional particle arrays for zions and electrons, respectively. The two arrays contain seven attributes for each particle, including coordinates, velocity, weight and particle ID. In a production run, each GTS process can generate up to 230MB of particle data per output.

The GTS particle data can be visualized by using parallel coordinates [64, 121]. Parallel coordinates is a visualization method commonly used to depict and analyze multivariate data. To generate parallel coordinates from GTS particle data, each

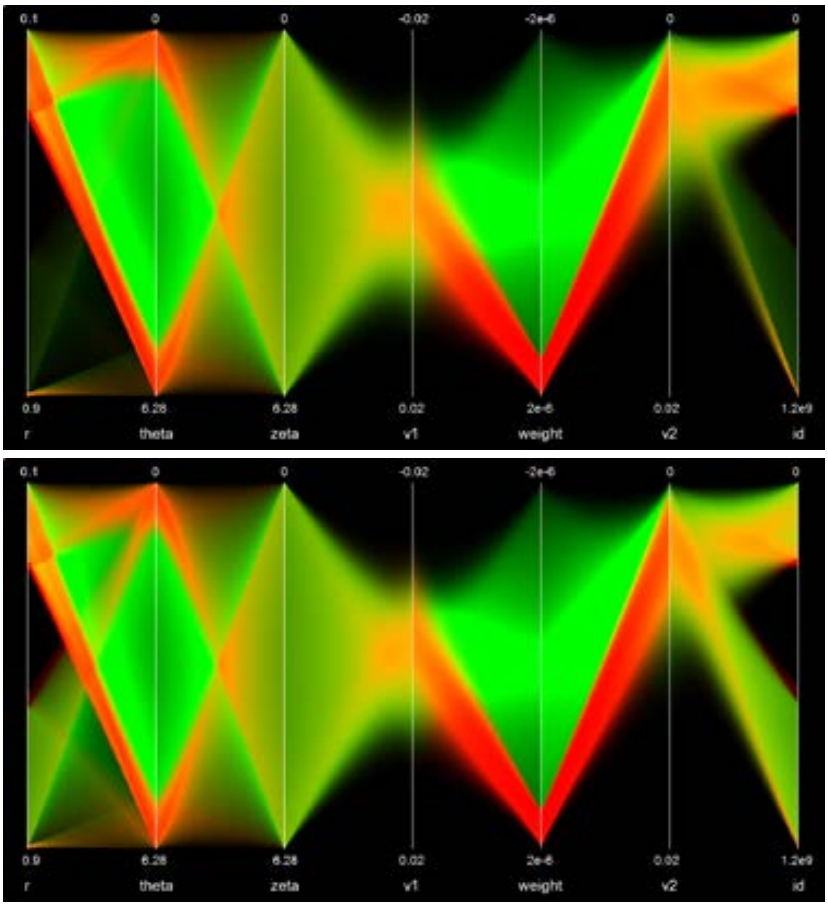


Figure 3: Parallel coordinates for GTS particle data. The two images are drawn from 2 timesteps of particle data each with 120GB in size. The red lines highlight particles with the absolute 20% largest weights.

processor first generates its local plot of parallel coordinates from the selected particles. Then, all processors collectively generate the final plot through parallel image composition [153]. Multiple plots of parallel coordinates can be generated and composed to show the relationship between different groups of particles. Figure 3 shows the parallel coordinates for two time steps, where the green areas correspond to all particles, and the red areas corresponds to the particles with the absolute 20% largest weights. Our parallel coordinate analytics can clearly show the evolution of particle data distribution at large scale.

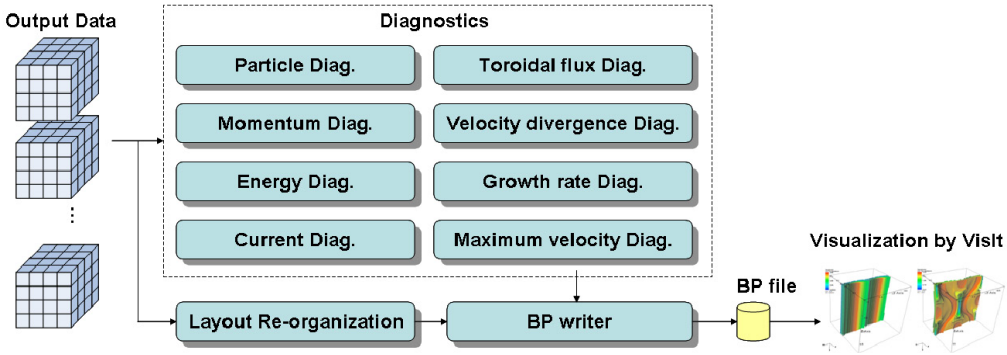


Figure 4: Illustration of online data analytics for Pixie3D.

2.1.3 Pixie3D

Pixie3D [30] is a 3-Dimensional MHD (Magneto Hydro-Dynamics) code that solves the extended MHD equations in 3D arbitrary geometries using fully implicit Newton-Krylov algorithms. Pixie3D employs multigrid methods in computation and adopts a 3D domain decomposition. The output data consists of eight 3D arrays that represent mass density, linear momentum components, vector potential components, and temperature, respectively.

As illustrated in Fig. 4, various diagnostic routines are performed on Pixie3D output data to generate derived quantities such as energy, flux, divergence, and maximum velocity. These derived quantities, along with the raw output data, are then read by visualization tools like VisIt for interactive visual data exploration. Pixie3D employs the BP file format for fast write performance [87]. Array layout re-organization is performed to speed up subsequent read access.

2.1.4 S3D

The fourth application is S3D combustion simulation code. S3D is a state-of-the-art flow solver for performing direct numerical simulation (DNS) of turbulent combustion [44]. During its execution, S3D simulation periodically outputs species data which are 22 3-dimensional double-typed arrays.

There are two types of visualization for the S3D data. The first is a parallel volume

rendering program [152] which renders images for each every species. The volume rendering program uses the same 3D domain decomposition as the S3D simulation, but may run on a different number of processes. Therefore, the 3D arrays need to be re-distributed from the simulation processes to the analytics processes. Such a data exchange pattern is so-called "MxN" [13] and requires appropriately

The second visualization is iso-surface extraction. An iso-surface is constructed by first finding all voxels in the volume data whose values contain a given iso-value, and then applying the classic Marching Cube algorithm [91] to generate polygons for the iso-surfaces. The iso-surface extraction can be accelerated via the Octree index [128]. We build an online Octree from the S3D volume data. During the Octree construction, we use a pre-defined iso-value range to guide the refinement of Octree, so that the regions which contain iso-values are indexed by Octree nodes at finer granularity. Each node in Octree contains a minimum and maximum value of its covered volume. To perform marching cube computation, the Octree is traversed from top down in breadth-first order, and the iso-value is used to filter out branches whose ranges do not to overlap with the iso-surface.

2.1.5 LAMMPS

LAMMPS (Large Scale Atomic/Molecular Massively Parallel Simulator) [114] is a popular Molecular Dynamics simulation code. It is written with MPI and performs force and energy calculations on discrete atomic particles. LAMMPS can be coupled with the SmartPointer [148] analytics pipeline for online data exploration. As part of the SmartPointer pipeline, the Bond analytics program takes as input the atoms array emitted from LAMMPS simulation, and calculates and outputs bonded atom pairs (two atoms whose distance is within a pre-defined threshold) among all atoms.

The original Bond implementation uses a two-level loop to calculate bonded atoms and has a complexity of $O(N^2)$ where N is the total number of atoms. RTree index

can be used to accelerate Bond computation. A RTree index can be built using the 3D coordinates of atoms. Then for each atom, a spherical query is performed on the RTree to find those atoms whose distance is within the threshold and hence is bonded with the query atom. Both RTree construction and query is of complexity $O(NlogN)$, so the total time complexity of Bond is reduced to $O(NlogN)$.

2.1.6 Summary of Application Requirements

From the application use cases described above, we can make the following observations regarding application's requirements for online data analytics supporting platforms.

Need for High Performance Data Movement. The online data analytics needs to consume live simulation output data at runtime. Since the simulations typically run at large scale and can generate massive volumes of data, the data movement between simulation and analytics can have a significant impact on overall performance. Therefore, the underlying data movement facility needs to achieve high throughput and low latency and keep its impact on application performance as small as possible.

Supporting Complex Data Exchange Patterns. In most applications, simulation and analytics are implemented as separate parallel codes. The data movement between them consists of a potentially large number of parallel data streams between multiple processes. Such data exchange can get further complicated by the fact that the data distribution and format at simulation side may differ from the one adopted by the analytics side (the 3D array data re-distribution in S3D is one such example). It would be very difficult and error-prone for domain scientists to implement such complex data exchange patterns by themselves. Therefore, certain high level abstraction is needed to ease the expression of data distribution at simulation and analytics sides and the data exchange between them. Such a high level interface should be

coupled with well-tuned underlying data transports to achieve high performance.

Need for Rich Meta Data. In order for online data analytics to process the simulation output data, sufficient information regarding the data need to be carried from simulation to analytics. Such meta-data ranges from variable name, data types and array shapes, to various data indices (e.g., the R-Tree and Octree spatial indices required by analytics for LAMMPS and S3D, respectively). The meta-data not only makes it feasible for analytics to interpret the data, but can also lead to performance improvements (e.g., using index to speed up queries on data); besides, the presence of meta-data makes data self-descriptive and avoids hard-coding analytics for specific data structure/format. Therefore, the underlying supporting platform should provide support for meta-data generation, dissemination and interpretation. Among the various types of meta-data, support for online spatial indices is particularly important due to their wide usage.

Supporting Diverse Analytics Computation Models. We define the computation model of an analytics as the order in which it processes input data and the way its computation is parallelized. The analytics used in the five applications are diverse in their computation models: some can process simulation output data in arbitrary order while others requires specific ordering of data; some are sequential programs, some are multi-threaded, while others are implemented as MPI parallel programs. Ideally, the underlying supporting platform should pose few restrictions on analytics computation model to allow generality and flexibility in analytics implementation.

2.2 Performance Model

In this section, we develop a simple analytical performance model for coupled simulation and analytics. The main purpose of the model is to capture the fundamental factors impacting the end-to-end performance and cost, so that we can quantitatively reason about the principal design trade-offs in online data analytics middleware. In

particular, it can help us to compare various existing designs listed in Section 1.1.2. Such comparison supplements other efforts [33] focused on the usage models and software engineering implications associated with different online data analytics middleware designs.

2.2.1 Performance and Cost Metrics

Before introducing our model, we first define a few performance and cost metrics.

Total Execution Time: the time from the start of simulation and analysis to the completion of both, also termed "Time to Insight" in our prior work [10]. This performance metric measures how long it takes to finish both the simulation and analysis, the idea being that it is the scientific insights derived from data analysis that drive science end users [15].

Total CPU Hours: the total nodes used multiplied by the total execution time (in units of hours). This metric measures the cost of a run, as supercomputing centers commonly charge end users with the total number of CPU hours consumed by their jobs. Another useful cost metric is a run's total power consumption [48], which we will consider in our future work.

2.2.2 Analytical Modeling

Our model is constructed by comparing the performance (Total Execution Time) and cost (Total CPU hours) with the Inline vs. Staging approaches, both of which have been actively developed and successfully applied in practice. In the Inline approach, analytics are performed by simulation processes synchronously in compute nodes, while in the Staging approach, simulation output data is transferred to a dedicated set of staging node and processed there by analytics. We will show how to extend the model to other cases in Section 1.1.2.

For modeling purpose, we consider the simple but common usage scenario of online data analytics (shown in Figure 5(a)). In this case, the simulation periodically

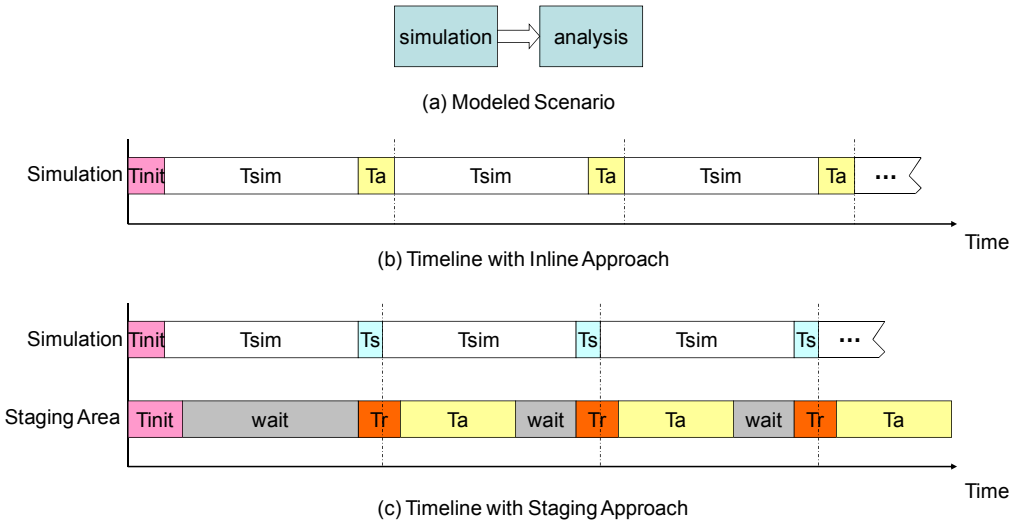


Figure 5: Timeline for Inline and Staging approaches.

generates output data and passes the data to an analysis component, which then immediately performs certain processing actions. The term "analytics" is used to denote actions ranging from simply writing data to storage, to data analytics such as feature extraction, indexing, compression, to the processing needed for coupling scientific codes, to data conversions for storage, and data visualization. We assume a processing model in which such actions are arranged as computational dataflow graphs, where each such directed graph describes the inputs/outputs of individual, indivisible analysis actions and the data movements between them. The formulation shown below may be applied to any bi-section cut across this graph to evaluate the placement of all computations before and/or after the cut. The notations used in our model are summarized in Table 1.

We denote the Total Execution Time using Inline and Staging approaches as *Tinline* and *Tstaging*, respectively. Figure 5(b) and 5(c) separately show the timeline of simulation and analysis with the Inline and Staging approaches. Omitting the initialization and finalization phases of long running simulation, *Tinline* and *Tstaging* can be calculated as follows:

Table 1: Major notations used in model.

Psim	Total number of nodes on which simulation is run
Pa	Total number of nodes in staging area (if present)
Tsim(P)	Simulation's wall-clock time between two consecutive I/O dumps when running on P nodes
Ta(P)	Analytics' wall-clock time for processing one simulation output step when running on P nodes
Ta(P)	Analytics' wall-clock time for processing one simulation output step when running on P nodes
K	Total number of I/O dumps
α	$\alpha = Pa/Psim$
β	$\alpha = Ta(Psim)/Tsim(Psim)$
Tsend	Simulation-side visible data movement time
Trecv	Staging node-side visible data movement time
s	Slowdown factor of simulation

$$Tinline = K \times [Tsim(Psim) + Ta(Psim)] \tag{1}$$

$$Tstaging = K \times max\{Tsim(Psim) \times s + Tsend, Trecv + Ta(Pa)\} \tag{2}$$

Define the performance speedup of using Staging over Inline:

$$Speedup = \frac{Tinline}{Tstaging} \tag{3}$$

And let $\alpha = Pa$ / $Psim$(size of staging area as percentage of total number of simulation nodes),and $\beta = Ta(Psim)/Tsim(Psim)$.This results in:

$$Speedup = \frac{Tsim(Psim)(1+\beta)}{max\{Tsim(Psim) \times s + Tsend, Trecv + Ta(Psim \times \alpha)\}} \tag{4}$$

An upper bound on Speedup can be derived as:

$$Speedup < (1+\beta)/s \tag{5}$$

In the formula above, β denotes analysis time as percentage of simulation time at the scale of *Psim* nodes, and s is the slowdown factor of simulation time due to staging (s$\geq$1) (e.g., slowdown due to network contention caused by additional data movements needed for staging [12], if any).

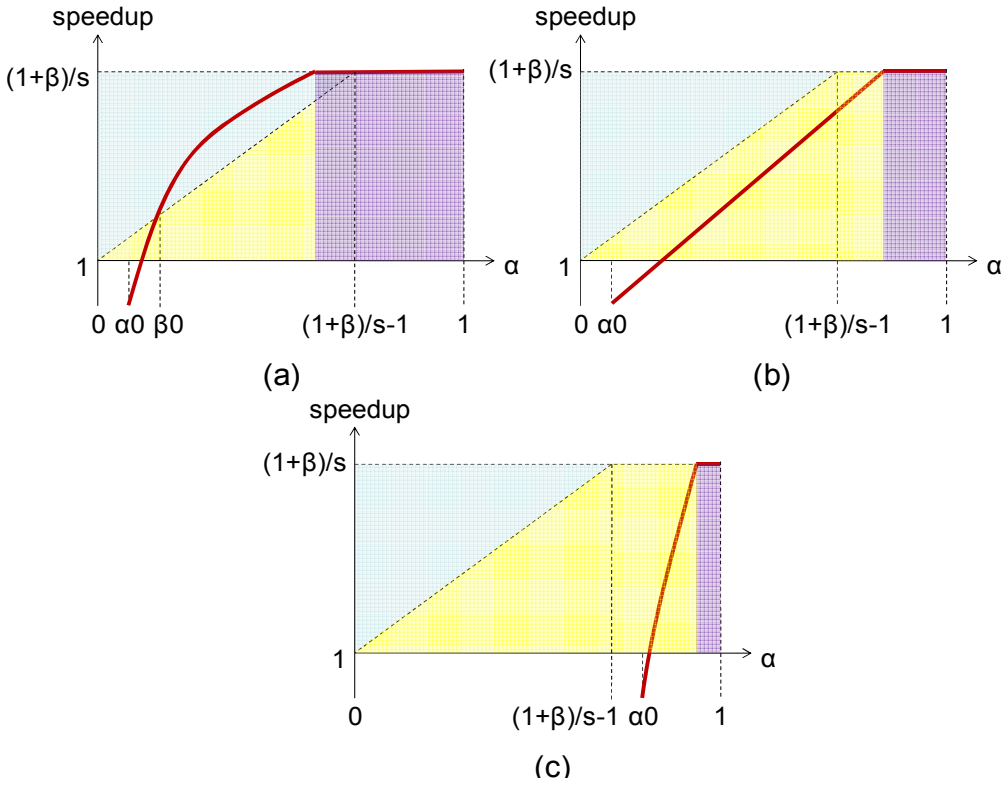


Figure 6: Speedup of Staging vs. Inline.

Since the Staging approach uses additional Pa nodes to offload analysis and may improve the total execution time through pipelining effect, it is interesting to understand the conditions under which staging can achieve the maximum speedup with some associated cost. Figure 6 shows three different possible relationships between staging area size (α) and speedup. In each figure, there are three regions: inefficient region (or sub-linear speedup region, colored yellow), efficient region (or super-linear speedup region, colored blue), and over-provisioned region (where no more speedup could be gained by increasing size of staging area, colored purple).

Figure 6(a) shows a case where Staging outperforms the Inline approach in both performance (speedup is greater than 1) and cost (*Total CPU Hours*). The conditions are: (i) no slowdown, i.e., slowdown factor s=1; (ii) no additional delay due to data movement to staging: $Tsend$=0; (iii) $Ta(P)$ scales sub-linearly with P; (iv) simulation time between successive output steps is larger than the time required to receiveand analyze data: $Tsim(Psim) > Trecv + Ta(Pa)$. Note that if analysis is sub-linear, then when scaling it down to run on some smaller number of nodes, the cost ($Ta(P) \times P$) is reduced. This may create a "sweet-spot" region, shown as [$\beta 0$, $(1+\beta)$/s-1] in Figure 6(a), where α% of additional nodes as staging area can speedup the total execution time by more than α%!

Figure 6(b) shows a case for linear-scalable analysis. As can be seen, if analysis scales linearly, i.e., can be performed locally on compute nodes with no communication, then there is no savings in CPU hours by offloading it to a staging area (since the product $Ta \times Pa$ is constant), but offloading will only introduce additional costs for data movement.

Figure 6(c) demonstrates a case where the minimum size of the staging area ($\alpha 0$), determined by the memory requirement to accommodate simulation output data plus the analysis code/data, is larger than $(1+\beta)/s$-1. In this case, the Staging approach with any staging area size α larger than $\alpha 0$ will always falls into the inefficient region.

The model can be extended to compare other approaches listed in Section 1.1.2. For example, we can simply set the *Tsend* and *Trecv* parameters according to file I/O performance to accommodate the Offline approach. For the Helper Core case, we can set the *Tsend* and *Trecv* parameters according to intra-node shared memory data movement performance, and set the slowdown factor s to capture interference effect between co-located simulation and analytics due to contention on shared on-node resources. The performance and cost with Hybrid approach can be modeled in a way similar to Formula (2) (that is, treat the coupled simulation and analytics as a pipeline), but takes into account that a portion of analytics computation takes place synchronously at simulation side. In fact, [63] derives a model for the Hybrid case exactly in this way and we refer interested reader to it.

2.2.3 Implications from the Model

We use the performance model to review various existing approaches to online data analytics mentioned in Section 1.1.2. It should be noted that those approaches mainly differ in where to place online data analytics along the I/O path. We make the following observations.

Firstly, the Staging approach can benefit non-scalable analysis actions. An interesting property of Staging is that the performance improvement is more evident with less scalable analysis. Our work with GTC, Pixie3D and Chimera applications [160, 11] evaluated the feasibility of offloading various operations to a separate staging area (e.g., file writing, format conversion, array layout re-organization, histogram calculation, indexing, and sorting), and achieved the "sweet-spot" region shown in Figure 6(a) for both applications at large scale. A similar conclusion can be made for the Helper Core approach as well.

Secondly, offloading linear-scalable analysis onto the staging area or helper cores is less cost-effective than placing it inline, since there will be additional costs for data

movement but no reductions in total CPU usage. Data filtering [104], sampling [10], in-situ compression [73] and visualization (such as slicing, iso-surface, and PCA) [102] fall into this category. Those operations can scale to large core counts, extract subsets or features of raw data and hence reduce data volume, and sometimes share large amount of input and/or meta-data with simulation, all of which makes it beneficial to place them inline with the simulation.

Thirdly, for approaches like Staging and Helper Core which perform analytics on separate resources and asynchronously with simulation, it is important to move data in a way such that (i) simulation-side visible data movement latency ($Tsend$) is minimized; (ii) the slowdown factor (s) is minimized; (iii) receiver-side data movement latency ($Trecv$) is reduced to leave sufficient time for analysis to complete before the next I/O action. Our work with the Staging approach show that it is feasible to meet those conditions in practice: (i) by using middleware that provides specialized data copying and marshaling mechanisms to achieve very low simulation-side visible data movement latency ($Tsend$) [10]; (ii) by using contention-aware scheduling for asynchronous data movement to mitigate the slowdown factor (s) [12]. As a forwarding note, this dissertation work will additionally leverage a chunk-based computation model to overlap receiver-side data movement latency ($Trecv$) with analysis computation and reduce the memory requirements of the staging area ($\alpha 0$) [160]. We will also show how to manage those factors to improve performance for the Helper Core and other setups.

Fourthly, the applicability of any approach is constrained by memory availability. For the Helper Core approach, the aggregated memory usage of co-located simulation and analytics plus memory cost for intra-node data movement can not exceed compute nodes' DRAM capacity. For the Staging approach, at simulation side, extra memory space is needed for asynchronous data movement; the staging area should contain sufficient nodes ($\alpha 0$) to accommodate the simulation output and all other data and code

to run analysis functions. The Inline approach seems to require less memory than others, but its applicability and benefit is constrained by other factors mentioned above. Potential solutions to the DRAM space constraint are i) implementing analytics in computation models which process data incrementally (e.g., one-pass streaming), and ii) extending DRAM capacity with alternative storage media like SSDs and running analytics in out-of-core fashion on the resulting deep memory hierarchies. We will explore both directions in this dissertation.

In summary, the major implications from our performance model are:

1) There is no single, the best analytics placement, and therefore analytics placement should be made flexible.

2) Analytics placement, data movement, interference between simulation and analytics, and memory efficiency of analytics are the factors which can significantly impact the end-to-end performance and cost of coupled simulation and online analytics.

2.3 Design Choices for Online Data Analytics Middleware

2.3.1 Limitations of Existing Solutions

According to the application requirements listed in Section 2.1.6 and the implications from the performance model in Section 2.2.3, the limitations of existing solutions are evident.

The common, major shortcoming is their lack of support for flexible analytics placement. Most existing systems support certain, fixed placement choices (including Inline, Helper Core, Staging, Active Storage, and Offline approches) and therefore, each is efficient or applicable to a limited classes of analytics. The Hybrid approach permits analytics to run at different locations, but require adopting particular coding patterns or re-placement involves substantial re-coding. Further, they do not support seamlessly switching analytics between online and offline, nor do they allow dynamic

changes in analytics placement. Such inflexibility in turn makes it very difficult (if not possible) to tune performance through changing analytics placement, even though the potential improvements can be high. Typical causes of limited flexibility are (1) an inability to handle the alternative data movements between simulation and analytics required by different placement options (i.e., supporting inter-node and intra-node data transfer and file I/O); (2) lack of uniform higher-level interfaces that hide data movement detail; (3) imposition of specific computation models for analytics; and (4) inability to achieve those requirements with high performance and scalability.

Besides, all existing solutions use dedicated resources to host online data analytics. As we will show in Chapter V, many simulations actually leave substantial amounts of unused idle resources (CPU cycles and memory space) in compute nodes during their execution. Those otherwise-wasted idle resources can be leveraged to run online analytics to achieve significantly better resource efficiency. However, doing so requires co-locating analytics and simulation on the same nodes and in turn cause severe interference to simulation due to contention on shared on-node resources (such as Last-Level Cache space and memory bandwidth). Existing solutions either disallow such co-location or are largely ignorant of the interference, resulting in sub-optimal performance and huge waste of resources at large scales.

Furthermore, existing solutions only provide limited meta-data support and generally do not allow extension or customization of meta-data. Such limitation often cause user to implement hard-coded, non-reusable analytics routines. Another severe consequence of limited meta-data support is that it often excludes the use of indexing techniques such as RTree and Octree indices required by LAMMPS and S3D, despite that fact that those indices can be very useful for accelerating many online data analytics.

2.3.2 Our Solution

From our discussion so far, it is clear that a "good" supporting platform for online data analytics should achieve the following design goals:

- Goal No. 1: Provide high level abstract interfaces to easily express complicated data exchange between simulation and analytics.
- Goal No. 2: Support high performance data movement between parallel simulation and analytics programs.
- Goal No. 3: Allow placement flexibility of data analytics along the I/O path.
- Goal No. 4: Make data analytics memory efficient.
- Goal No. 5: Effectively control interference between simulation and analytics.
- Goal No. 6: Support rich meta-data including online spatial indices.

In order to overcome the limitations of existing solutions and better support online scientific data analytics, we develop middleware-level solutions by following the design goals.

To achieve the Goals No. 1, 2, and 3 (high performance data movement and analytics placement flexibility), we explore two different approaches: one from the programming model perspective, and the other from the I/O middleware perspective. The first approach, PreDatA (Chapter III), targets an important class of online data analytics termed "Preparatory Data Analytics" and provides a streaming, MapReduce-like programming model to decompose analytics into several stages and deploy them onto different resources along the I/O path to achieve high performance and scalability. The second approach, FlexIO (Chapter IV), is an I/O middleware to support diverse data exchange patterns and location-flexible online data analytics. FlexIO provides high performance, memory efficient intra- and inter-node data movement transports which allow diverse analytics placement options. It offers simple

high-level programming interfaces which can be used to implement complicated data exchange patterns and makes changes in analytics placement transparent to simulation and analytics codes. Various placement policies can be built on top of FlexIO to exploit the location flexibility for tuning application performance, CPU usage, and data movement cost. Comparing to PreDatA, the FlexIO approach further relaxes the computation model constraints and thus is more general. Nevertheless, both approaches are shown to be applicable to a broad range of useful analytics and achieve high performance at large scales.

Regarding the Goals No. 4 and 5 (resource efficiency and interference mitigation), we propose a runtime approach called "GoldRush" (Chapter V) which can run analytics using idle resources "stolen" from simulation on compute nodes without slowing down simulation performance. GoldRush applies fine-grained scheduling to harvest idle resources from the simulation in ways that incur negligible runtime overheads and minimize interference between the simulation and analytics. This in turn can significantly improve resource efficiency and reduce data movement costs with negligible performance impact or even improved end-to-end performance for some cases.

To achieve Goal No. 6 (rich meta-data and efficient spatial indices), we propose a general framework called "ZStore" (Chapter VI) which provides utilities for constructing and querying online spatial indices in a scalable fashion. ZStore provides a flexible in-transit workflow embedded in the I/O path for constructing index from live simulation output data in a streaming manner. It allows application-specific control over data distribution. Representative spatial indices such as RTree and Octree can be built with ZStore for online usage. ZStore is also optimized for deep memory hierarchies with SSD equipped, and therefore is well suited for out-of-core data exploration.

2.4 Conclusions

In this chapter, we study the application requirements for online data analytics supporting platforms using real-world use cases. We also develop a performance model for coupled simulation and online data analytics. Based on the application requirements and performance model, we conclude that existing solutions to online data analytics are insufficient in supporting high performance data movement, rich metadata (including spatial indices), diverse data exchange patterns, flexibility of analytics placement, and effective control of interference between simulation and analytics. This motivates us to develop new middleware solution to overcome those limitations and better support online data analytics.

CHAPTER III

ENABLING PREPARATORY DATA ANALYTICS

In this chapter, we target a common class of online data analytics termed "Preparatory Data Analytics", and present the PreDatA system to support that. We demonstrate the scalability and performance benefits of PreDatA with two real-world applications on up to 16384 cores.

3.1 Introduction

Scientific applications running on High End Computing (HEC) platforms can generate large volumes of output. As these grow to peta-scale and beyond, fast write and read accesses to massive data are becoming increasingly important, both to speed up the simulation and to accelerate exploration of data. A prerequisite to data exploration is that data is prepared in terms of data layout, indexing, and annotation. For example, some analysis tools prefer data to be laid out as contiguous arrays for quick loading [150], and queries can be accelerated if data is properly sorted and indexed [130]. In other words, appropriate data preparation is critical for data analytics, inspection, or visualization to operate efficiently. Finally, 'hidden' in the voluminous data sets generated by running simulations are latent data characteristics of interest to end users, an example being statistical measures that can be used to validate the veracity of the ongoing simulation, gain understanding of the simulation progress, and potentially, take early action when the simulation operates improperly [54].

The object of this chapter is the development of efficient methods that properly prepare data for subsequent inspection, storage, analytics, and even for input into concurrent, coupled simulation models (e.g., as in climate modeling). Our approach associates such data preparation with the output actions taken by simulations in ways

that speed up output actions, thereby also improving simulation performance. The software artifact developed and used for these purposes is the PreDatA middleware. PreDatA provides scalable and flexible ways of associating data preparation operations with the I/O actions of HEC applications, by generalizing the I/O stack used by HEC codes and taking advantage of the ADIOS I/O library [87] used in a wide variety of peta-scale codes. The enhanced I/O stack enables efficient operations on output data via predefined or user-provided computational functions. These functions are performed while I/O is ongoing by staging data to where PreDatA can leverage the computational power of selected machine nodes supporting I/O and/or connected to the storage subsystems. Further, by using PreDatA to index or properly annotate data, a reduction in the volume of subsequent reads performed by scientific workflows engaged in data analysis can be achieved. This also reduces interference at the parallel file system due to simultaneous writes used by output and reads used by scientific workflows.

The PreDatA middleware exploits the additional computational and memory resources provided by a staging area resident on the peta-scale machine. Output data are moved from compute to staging area nodes asynchronously to reduce write latency. PreDatA operations are applied to data prior to leaving the compute node and/or on data buffered in the staging area. The middleware provides a pluggable framework for executing user-defined operations such as data re-organization, real-time data characterization, filtering and reduction, and select analysis (or pre-analysis). These operations are specified in ways natural to the 'streaming' context in which they are used. Despite this rich functionality, PreDatA offers levels of performance not provided by current file system-based approaches to analyzing output data, as shown with extensive experiments in this chapter.

PreDatA performance is evaluated with several production peta-scale applications on Oak Ridge National Laboratory's Leadership Computing Facility platform. For

one application, GTC [70], at the scale of 16,384 compute cores and with 1.5% additional resource usage, PreDatA hides write latency by up to 99.9%, improves total simulation time by 2.7%, and achieves a 1.5% saving in total CPU usage compared with performing pre-analytics in the compute nodes. In this experiment, PreDatA generates scientifically meaningful statistics from the 260GB output data in one simulation time step in about 40 seconds. For another application, Pixie3D [30], using PreDatA to re-organize the array layout of output data from 16,384 cores improves subsequent read performance for these output files by 10 times compared to when no such reorganization is performed. At the same time, total execution time of the simulation is improved by 1% with only 0.7% additional resource usage.

The remainder of the chapter is organized as follows. Section 3.2 presetns background and motivation. Sections 3.3 and 3.4 present the design and implementation of PreDatA, respectively. Section 3.5 applies the PreDatA approach to the two applications, and evaluates the resulting performance demonstrating its advantage over other online and/or offline approaches. Section 3.6 summarizes related work, and Section 3.7 concludes the chapter.

3.2 Motivation

Conventionally, data preparation and analytics are performed either in compute nodes where the simulation is running or offline:

In-Compute-Node approach: operations are performed in the compute nodes where output data is generated. The processed output is then written to the parallel file system.

Offline approach: the simulation dumps data to a parallel file system. Analysis codes running on other resources read such data and operate on it.

These two approaches to processing simulation output data differ in terms of their respective costs and limitations. For the In-Compute-Node approach, the overhead

of data processing operations is visible to the simulation with consequent expenses in terms of CPU hours at scale. Performance advantages result if In-Compute-Node actions reduce output volumes, but severe performance penalties arise if data processing operations do not scale with the simulation. For the Offline approach, if the data volume is large, intermediate files may consume considerable storage resources, and parallel file system write and read times can be dominant causing high latencies and unacceptable levels of perturbation of peak file system performance. Therefore, it is clear that additional methods are needed to satisfy the I/O and data processing needs of the two representative peta-scale codes mentioned above.

One such method is the *Staging Area* approach. In this approach, a reasonable number of compute nodes are reserved as a Staging Area for staging data and hosting operations to apply to staged data before it reaches storage. Asynchronous execution within the Staging Area hides the processing costs from the simulation and affords an opportunity to employ less scalable operations 'at scale' since the Staging Area is small in comparison to the number of compute nodes being used (e.g., using a ratio of 128:1 for compute cores to staging cores). It is also possible to reduce disk accesses by pre-processing data so as to permit later analytics to focus on the data that is most relevant. Using these insights, the PreDatA middleware exploits the benefits of the Staging Area approach.

3.3 PreDatA Middleware Design

The PreDatA middleware design augments the current I/O stack on HEC platforms with data staging and in-transit processing capabilities by exploiting computational resources in both compute nodes and the staging area for preparatory data analytics.

As shown in Figure 7, the PreDatA middleware resides in both the compute nodes on which the application runs and the staging nodes. Operations can be hosted in either location. When the application performs I/O actions, PreDatA acquires output

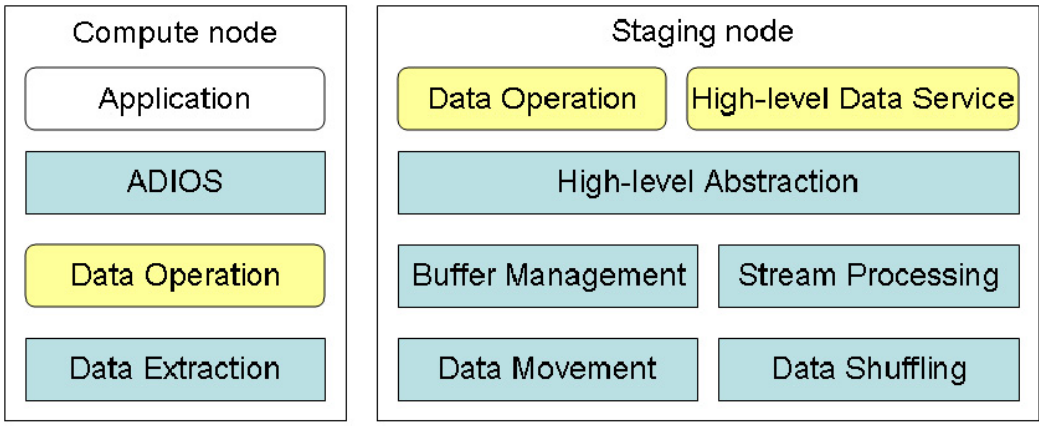


Figure 7: PreDatA middleware architecture.

data through the ADIOS I/O interface [85], stages data from compute nodes to staging nodes and performs in-transit data processing along the data flow.

There are several key features of PreDatA:

Asynchronous data movement. Data movement from compute to staging nodes is performed asynchronously to hide write latency from the simulation at a moderate cost of data buffering in the compute nodes. PreDatA explicitly schedules such asynchronous data movement to minimize interference with the simulation's communications.

Pluggable pre-data analytics. PreDatA provides a pluggable framework making it straightforward for end users to specify, deploy, and debug data processing operations. The programming interface is general enough to implement a variety of operations, including data re-organization, real-time data characterization, filtering and reduction, and lightweight data analysis.

User-defined operations. The middleware supports user-defined data operations with common services for data access, buffer management, scheduling and executing data processing actions, and high performance data exchange and synchronization across staging nodes.

Higher-level Data Services. The middleware also provides supports for building

higher-level data services ranging from data indexing and query to inter-application data exchange.

Integrated operations, separated from application codes. PreDatA hides from data processing codes the complexities of data access in the staging area while meantime offering high performance through permitting such codes to directly access buffered data. I/O stack integration is performed so as to separate application codes from the potential complexities of data processing actions.

3.4 PreDatA Middleware Implementation

The PreDatA middleware's implementation leverages our earlier work [12] on efficiently scheduling data movement from compute nodes to the Staging Area. The EVPath [45] high performance event system is used for efficient data buffering and manipulation in the Staging Area. The FFS [46] binary data encoding facility is used for in-transit data to provide PreDatA operations access to buffered data with rich meta-data information. The ADIOS [85] library is the basis for integrating PreDatA with application I/O.

3.4.1 Data Extraction and Movement

PreDatA uses the ADIOS I/O library as the basis for both the simulation's I/O stack and for PreDatA operations to access data output by the simulation. ADIOS allows for introducing PreDatA processing into the compute nodes without requiring changes to application codes, thereby insulating application code from the complexities of additional processing actions in the I/O stack. ADIOS also explicitly defines the structure of application's output data, and such meta-data information is used as a common interface for application and PreDatA operations to coordinate sharing data.

PreDatA also uses the scheduled, asynchronous RDMA [25] operations explained in [12] for extracting and moving data from compute nodes to staging nodes. The use of asynchronous RDMA reduces the write latency visible at compute nodes and

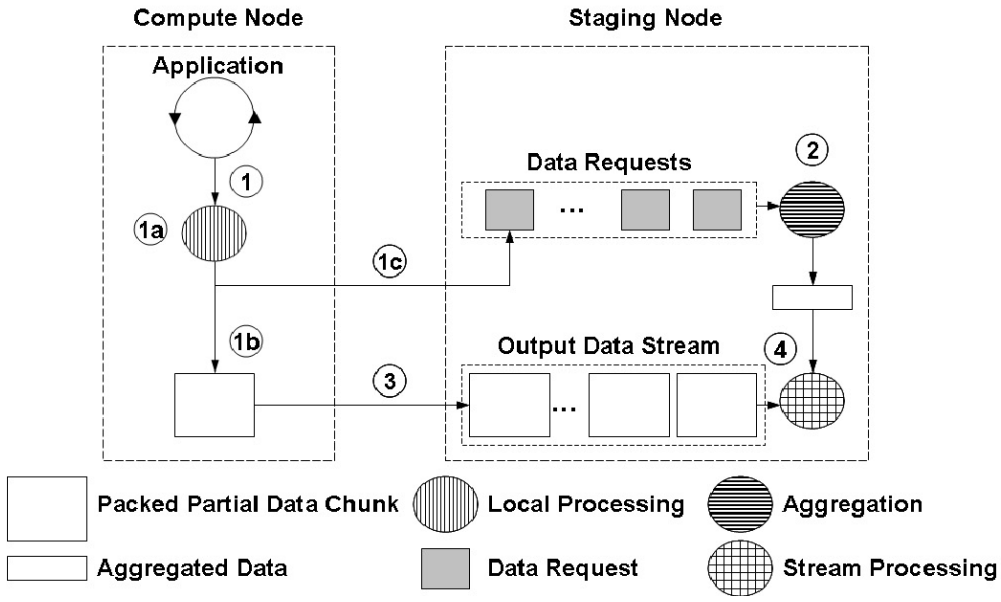


Figure 8: Overall data flow of PreDatA.

scheduling such RDMA operations helps minimize interference between communications performed by the simulation vs. those used for output. This is particularly important when output data movement overlaps collective communications among compute nodes and thereby may cause severe perturbation on simulation performance.

3.4.2 In-transit Data Processing along Data Flow

PreDatA augments the I/O stack resulting in the overall data flow shown in Figure 8. There are four stages in the data flow: (1) data extraction and optional local processing in compute nodes, (2) optional aggregation in staging nodes, (3) asynchronous data movement from compute nodes to staging nodes, and (4) data stream processing in staging nodes.

When I/O is triggered in the compute nodes, output data is passed to the PreDatA runtime in the compute nodes via the ADIOS interface (shown as Stage 1 in Figure 8). Typical output data of compute nodes consists of one or more scalars, local arrays, and/or partial chunks of global arrays. PreDatA executes a user-defined routine, if

provided, on the local output data (shown as Stage 1a in Figure 8). This constitutes a optional first pass of processing on the output. Possible operations include generating meta-data such as array dimension information, calculating local min/max values of partial array chunks, and filtering out undesired regions. All output data (scalars, local arrays, partial chunk of global arrays) are then packed into a contiguous buffer, termed a *packed partial data chunk*, using the FFS [46] binary data encoding facility (shown as Stage 1b in Figure 8). The structure of each packed partial data chunk is compatible with the ADIOS output data group definition, and metadata about the data structure is embedded in the packed partial data chunk. A data fetch request is sent to the staging node chosen by a user-overridable function *Route()* (shown as Stage 1c in Figure 8). PreDatA provides an interface that permits the data operation in Stage 1a to attach small partial results to data fetch requests, allowing for additional flexibility in the staging area. The compute node then resumes computation while the data movement and operations are performed.

In the Staging Area, each staging node waits for data fetch requests from compute nodes. When the staging node finishes gathering requests from all compute nodes it serves, it extracts partial results attached to requests, if there are any, and performs user-defined aggregation functions on them to generate aggregated results such as global array size and offsets, prefix sum, and global min/max values (shown as Stage 2 in Figure 8). Each staging node then begins to fetch packed partial data chunks from compute nodes (shown as Stage 3 in Figure 8). Data chunks are processed by staging nodes one by one in a streaming manner (shown as Stage 4 in Figure 8) and the aggregated results generated in Stage 2 are accessible from the stream processing operations.

In summary, the PreDatA middleware provides two passes across an application's output data. The first pass optionally done on compute nodes is suitable for operations that do not require global communications and/or synchronization. The second

pass performed on staging nodes, in a data streaming fashion, can be used to compute global data properties and/or to reorganize data for later storage. Data streaming is critical because it is unlikely for staging nodes to have sufficient memory to hold all of the raw data generated by multiple and, often, even single simulation output steps.

3.4.3 Stream Processing in the Staging Area

As mentioned above, the output data of each compute node is packed into a contiguous memory buffer, i.e., a *packed partial data chunk* and moved in its entirety into the Staging Area. From the Staging Area's perspective, incoming data consists of a finite number of packed partial data chunks streamed from compute nodes participating in the I/O dump. When there are multiple staging nodes, the packed partial data chunks are split into multiple streams across these nodes.

Each staging node is responsible for processing a stream of packed partial data chunks with each chunk from one compute process, which is the forth stage of the dataflow as shown in Figure 8. The processing of such a stream is divided into five phases (as shown in Figure 9):

Initialize: the *Initialize()* function of each operation is executed once at the beginning of an I/O dump with aggregated result data generated from the pre-fetch process (as shown in Figure 8) as a parameter to initialize the operation-specific data structure and for other setup tasks.

Map: the *Map()* function of each operation is executed on each packed partial data chunk. Intermediate results are tagged and stored in a local buffer.

Shuffle: when the last chunk within the I/O dump is processed, partial results are combined locally, if the *Combine()* function is provided. Each staging node applies the *Partition()* function to route intermediate result to other staging nodes according to the associated tag.

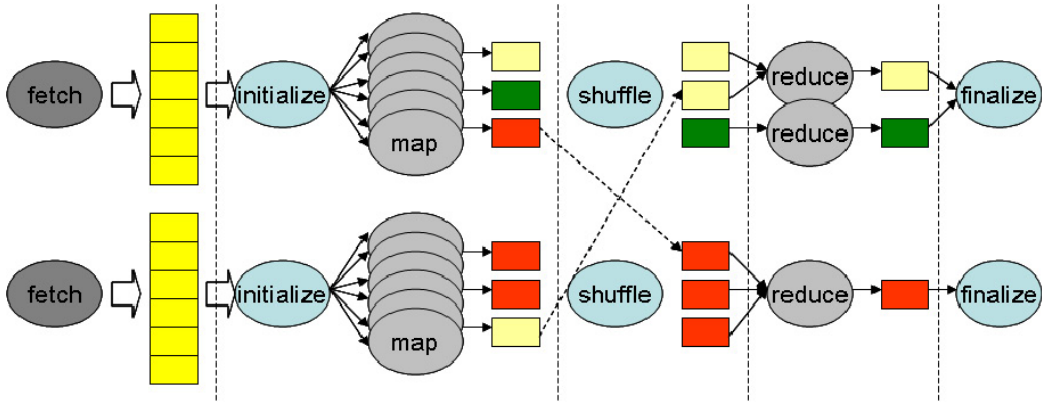


Figure 9: Stream processing in the Staging Area.

Reduce: each staging node groups intermediate results, both local and those received from other staging nodes, by associated tags and then performs the *Reduce()* function on each group of intermediate results to aggregate results.

Finalize: when the Reduce phase finishes, each staging node executes the *Finalize()* function of each operation, which writes final results to disk, feeds data to other consumers, and/or performs necessary cleanup.

Note that this data processing model is similar to the MapReduce [38] paradigm, with four notable differences: (1) the data processing model requires that the operations only need to read data once so that data can be processed in a streaming fashion, (2) the addition of the initialize and finalize phases, (3) users can can customize the data shuffling with highly-optimized MPI routines, (4) there is no central master that has global knowledge of data location and task progress.

A user can plug their own data operations into PreDatA middleware by implementing the functions mentioned above. They may also customize data movement scheduling policy to place data chunks within the data stream into specific order (e.g., fetching chunks in order of compute nodes' MPI rank for calculating prefix sum).

The staging area is running as a separate MPI program launched independently with the simulation. Each MPI process runs on one staging node. Within each staging

node, there are multiple threads in each MPI process that execute different pieces of the execution flow shown in Figure 9 to exploit concurrency.

3.4.4 Buffer Management

On compute nodes, additional buffering is needed to hold packed partial data chunks with a buffer size roughly equal to the output data sizes and configurable through the ADIOS configuration file. On staging nodes, all incoming packed partial data chunks are stored in buffers provided by the PreDatA runtime. The runtime maintains reference counts for recycling a buffer when the input chunk has been processed by all operations. For intermediate data received from other staging nodes during shuffling, data operation routines indicate to the runtime system when to recycle those buffers. Private buffers maintained by individual operations are its own responsibility. The latter is consistent with a basic assumption about the staging area made by the PreDatA middleware, which is that all data is maintained in in-core buffers. This means that for extremely large datasets, it is the responsibility of specific PreDatA operations to be aware of and deal with memory limitations. For assistance, PreDatA provides explicit memory manipulation routines that retrieve information about available memory space and allocate/de-allocate buffer space. The in-core assumption is reasonable for our target application workloads and platform, since there are no local hard disks or Solid State Drives (SSD) [109] attached to staging nodes in the tested environment. If such were present or if there were fast access to a shared parallel file system as an external buffer without concerns about perturbing output performance [86], buffer management should be extended to include out-of-core functionality.

3.4.5 The DataSpaces Global Data Knowledge Service

The purpose of this section is to show that the 'in-transit' and 'online' approach of data output and manipulation used in PreDatA can be used to implement the

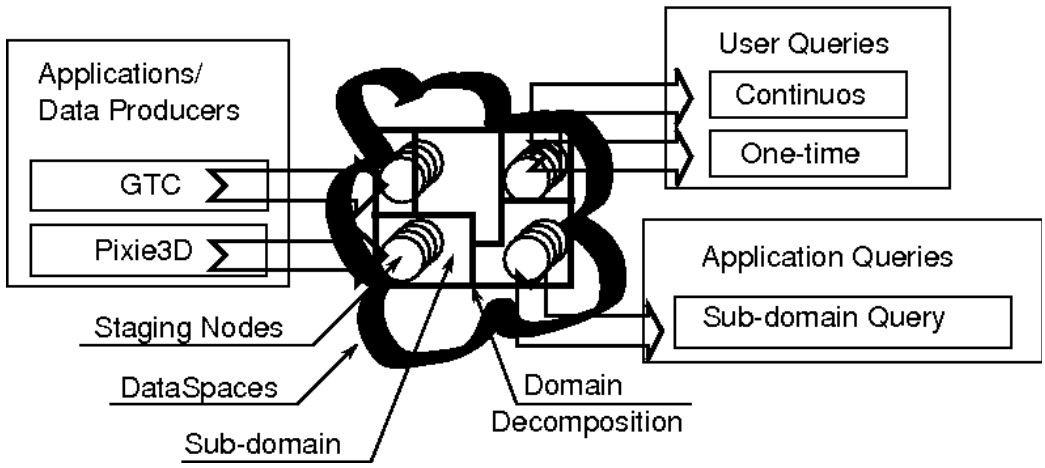


Figure 10: Example of application to application coupling implemented using the querying framework.

model-to-model communications used in high performance coupled codes [13, 157]. Toward that end, we are integrating into PreDatA the high level 'DataSpaces' data indexing and query services. The intent is to demostrate the extent to which pre-data analytics can be enriched to also support the rich and flexible methods for online access to generated data required for general inter-application interactions. DataSpaces provides higher level programmable and managed services for (1) data sharing – between operations working on a common set of data; (2) data redistribution – between operations with different data discretization and running on a different number of processors; (3) data indexing – data hashing for fast access; and (4) data querying – application data retrieval based on custom selectors. With (1)–(4), it provides the abstraction of a virtual semantically-specialized shared data space that can be asynchronously and flexibly accessed using simple yet powerful operators (e.g., *put()* and *get()*) that are agnostic of the location or distribution of data.

DataSpaces incorporates flexible mechanisms that can fetch and index data, on-the-fly, from multiple different sources, as shown in Figure 10. It can even extract data directly from a running application. It can store incoming data locally in the staging area or share it with the collaborating frameworks, index it for fast access, and

serve it in response to logged or incoming user queries. Datasets composed of both, homogeneous data types, e.g., doubles, floats or integers, as well as heterogeneous data types, e.g., aggregate structures of doubles, floats or integers, are supported.

DataSpaces implements a flexible querying mechanism that allows applications to request individual values as well as contiguous regions of data based on simple descriptors that are semantically meaningful to the application. For example, in the case of typical simulation data, data can be indexed based on its geometric coordinates within the multi-dimensional discretization used by the simulation allowing it to be queried using geometric descriptors that are meaningful to the application. Queries may be generated by users or by other applications. For example, each instance of a distributed querying application running on multiple nodes can query distinct and relevant sub-regions of data as needed. Similarly, a user can query sub-regions of interest only when they are needed or can register sub-regions of interest for continuous querying. In the latter case, for example, the user is notified automatically every time new data items that lie within the regions of interest are inserted into the space.

DataSpaces also supports aggregation and reduction queries. For example, queries can request the maximum or minimum value for a particular field in a given sub-region, or the average value of a specified field within a given region. Note that, from the perspective of a querying end user or application, the querying and data transfer process is transparent and independent of data distribution, i.e., the data comprising the query response may come from different nodes of the application that generated the data and served by different DataSpaces framework nodes.

DataSpaces complements the indexing and querying services with an in-memory data storage service. The storage service can be used to maintain private copies of the data extracted directly from a running application or store shared copies of the data processed by collaborating frameworks. The storage service incorporates a data coherency protocol that manages interactions with the data and ensures data integrity

when multiple entities simultaneously query the data.

DataSpaces maintains load balancing at two levels. First, the storage service distributes the data evenly across the DataSpaces nodes, and second, the indexing service dynamically distributes the index metadata across the DataSpaces nodes to distribute incoming queries across these nodes.

3.5 Performance Evaluation

As described above, the placement of operations can greatly affect their performance, the timeliness of the output, and impact the overall system performance. By evaluating several different operators using different placement choices, the benefits of the flexible placement and in-transit processing is demonstrated. The two driver applications mentioned in Section 3.2 are used to evaluate the PreDatA middleware and higher level DataSpaces services. Sorting, histogram, and 2D histogram operations are tested for GTC. The sorted particle data are written into BP files from the Staging Area. For Pixie3D, an array layout re-organization operation is created. This operation merges partial array chunks into larger contiguous ones for each of the eight 3-Dimensional arrays in Pixie3D's output and writes merged arrays to a BP file. Performance of DataSpaces global data knowledge service is also evaluated with GTC to demonstrate the feasibility of building high-level data services with PreDatA.

3.5.1 Experimental Environment

Experiments are run on Oak Ridge National Laboratory's Cray XT4/XT5 Jaguar platform. The XT5 partition contains 18,688 compute nodes in addition to dedicated login/service nodes. Each compute node contains two quad-core AMD Opteron 2356 (Barcelona) processors running at 2.3 GHz, 16GB of DDR2-800 memory, and a SeaStar 2+ router. The resulting partition contains 149,504 processing cores, more than 300TB of memory, over 6 PB of disk space, and a peak performance of 1.38

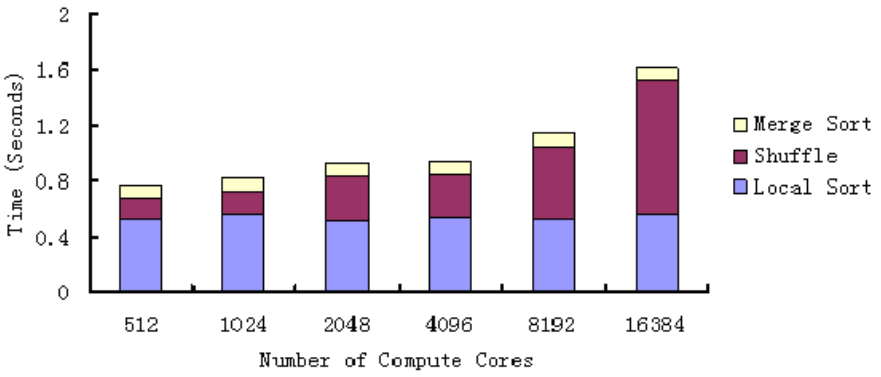
petaflop/s. The XT4 partition contains 7,832 compute nodes in addition to dedicated login/service nodes. Each compute node contains a quad-core AMD Opteron 1354 (Budapest) processor running at 2.1 GHz, 8 GB of DDR2-800 memory, and a SeaStar2 router. The resulting partition contains 31,328 processing cores, more than 62 TB of memory, over 600 TB of disk space, and a peak performance of 263 teraflop/s. For each case described below, we run each test case 5 times and use the best samples in both In-Compute-Node and Staging configuration for plotting to control for interference in the shared experimental environment.

3.5.2 GTC Performance

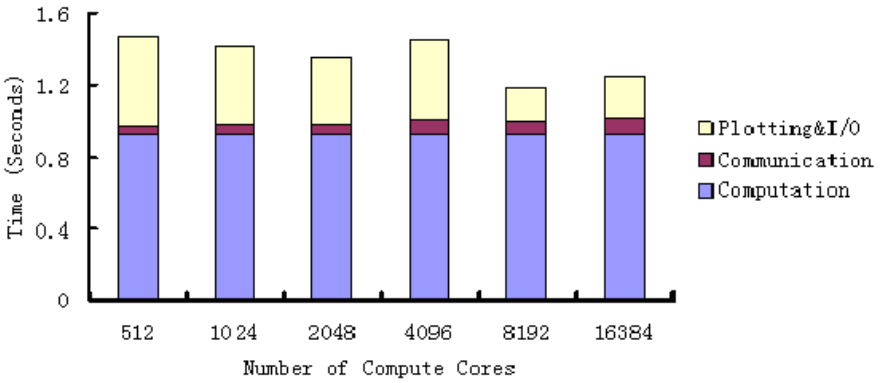
The GTC experiments are performed on the XT5 partition of Jaguar. As is typical with a production run, the GTC jobs are configured to deploy a single MPI process per node that spawns 8 OpenMP worker threads, one per core. I/O is only performed by the MPI processes. For GTC, three operations are tested: particle sorting, histogram generation, and 2D histogram generation. Each of these operators is applied to both the electron and ion particle arrays output with I/O interval of about every 120 seconds. Weak scaling is employed with 132MB total written per process for the two particle arrays. The Staging Area is configured to deploy 2 MPI processes per node with 4 worker threads per MPI process. The size of the Staging Area is adjusted to maintain a ratio of compute cores to staging cores of 64:1 (1.5%). That is, for each 64 nodes with compute processes (512 OpenMP worker threads), 1 node (2 staging processes for a total of 8 worker threads) is employed for staging. The tests are performed in two ways. First, all operations are performed in compute nodes and use synchronous MPI-I/O to write results ('In-Compute-Node' configuration). Then they are performed in Staging Area ('Staging' configuration).

3.5.2.1 Performance of Individual Operations

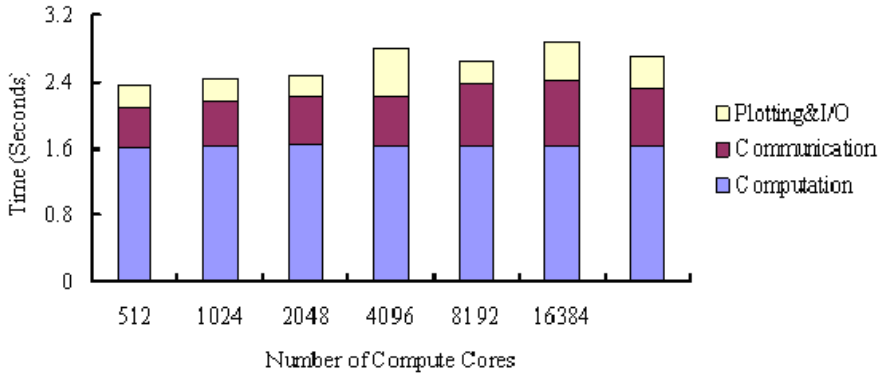
In this section we study the performance results for each operation.



(a) Sorting in Compute Node

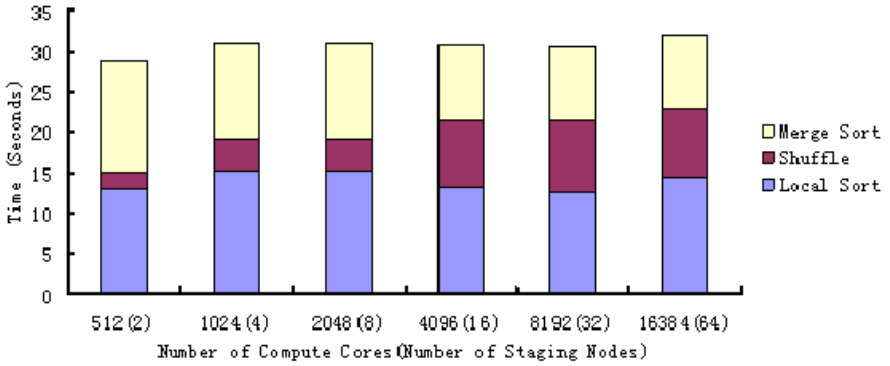


(b) Histogram in Compute Node

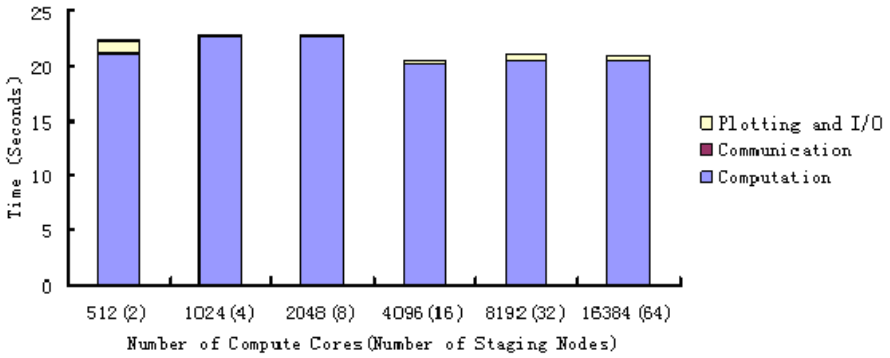


(c) 2D Histogram in Compute Node

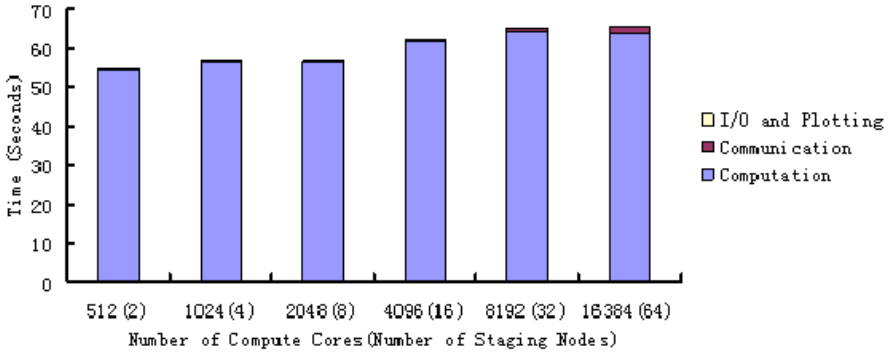
Figure 11: Timing results for individual operations Running in Compute Node.



(a) Sorting in Staging Area



(b) Histogram in Staging Area



(c) 2D Histogram in Staging Area

Figure 12: Timing results for individual operations running in Staging Area.

Sorting Operation: Figure 11(a) and 12(a) compare the performance of sorting using the In-Compute-Node configuration and the Staging configuration. Sorting is an example of communication intensive operations because it involves all-to-all communication and has minimal computational demands. When sorting in compute nodes, the data shuffle time among compute nodes increases dramatically as the operation scales and such cost is visible to simulation. On the other hand, sorting in the Staging Area takes at most 33 seconds at all scales, which is much less than the 120-second I/O interval. Therefore, performing sorting operation in Staging Area can mask the cost of sorting from simulation because of asynchrony. There are, however, 30 seconds of latency in Staging configuration, two orders of magnitude longer than the In-Compute-Node configuration. This tradeoff demonstrates the importance of placement: if the goal is to optimize simulation time, placing the sorting operation in Staging Area is better, but if the latency of generating sorted data is more critical, placing the operator in compute nodes is a better choice.

Histogram Operation: As shown in Figures 11(b) and 12(b), the histogram operation is computation dominant with communication contributing only a very small portion of the total operation time. While performing this style of computation intensive operation in the compute nodes takes less wall clock time, the perturbations to the total simulation time can be much larger due to the impact of I/O operation for saving histogram results. The time for writing the 8 MB histogram files ranges from 0.25 seconds to 7 seconds, which adds to the total simulation time. This reveals a different advantage for the Staging configuration: insulating simulation from variation in file system performance. Since the increased cost of computing the histogram is hidden by the asynchronous data transfer and operation savings, using the Staging configuration is still generally advantageous. For those cases where one has computation-intensive operations without a subsequent I/O operation or if latency is very important, using the 'In-Compute-Node' configuration is superior.

2D Histogram Operation: Similar to the Histogram operation, the 2D Histogram operation is computation dominant, as shown in Figures 11(c) and 12(c). While the computation and communication requirements for generating the 2D histograms is larger, the same conclusions can be drawn.

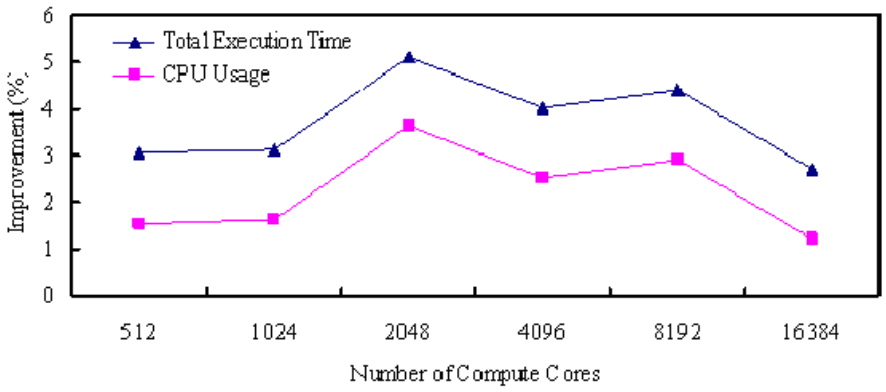
In summary, the results shown in this section demonstrate that for operations with different computation and communication characteristics, offloading operations from compute nodes to staging nodes generally helps mask the cost and variation of operation and associated I/O activity from simulation because of asynchrony but introduces longer latency for operation to finish because of the capacity mismatch between compute nodes and staging nodes. Depending on the latency requirements and variability in the system, performing these operations in a Staging configuration can contribute a performance improvement for some operations and insulation from system variability for others. In both cases, strict or weak latency requirements can override a short-term cost for an overall benefit.

3.5.2.2 Simulation Performance

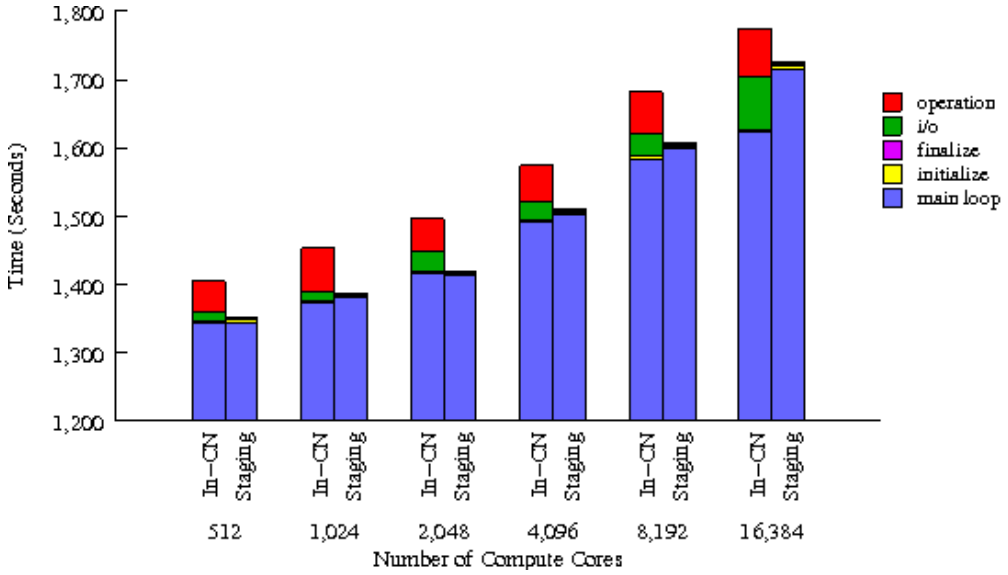
This section evaluates the GTC simulation performance in two different configuration. Figure 13(b) shows the total execution time of the GTC simulation for the two different configurations at various scales ranging from 512 to 16,384 compute cores. The Staging configuration improves the simulation's total execution time by 2.7% to 5.1% over the In-Compute-Node configuration (as shown in Figure 13(a)).

The breakdown of total execution time (shown in Figure 13(b)) explains the performance advantage of the Staging over the In-Compute-Node approach:

Firstly, the Staging approach hides write latency via asynchronous data movement. For example, at the scale of 16,384 compute cores, 8.6 seconds are required, on average, to write 260GB of particle data with the ADIOS synchronous MPI-I/O method. The visible I/O blocking time with the Staging configuration is reduced to



(a) Improvement of GTC simulation performance and cost



(b) GTC total execution time breakdown

Figure 13: GTC simulation performance.

0.30 seconds on average. This improvement of write latency increases with simulation scale.

Secondly, the Staging approach also insulates the simulation from the increasing time costs for performing the operations as the simulation scales since these operations are done in the Staging Area concurrently and asynchronously with the simulation. For the In-Compute-Node configuration, the time spent in operations increases from 3.0% to 4.1% as the simulation scales from 512 to 16,384 cores. With the Staging approach, the simulation spends no time carrying out such operations. While it is true that the Staging Area experiences a larger proportional time in performing the operations, the time insulating effects of asynchronous I/O afford using more time without impacting the application wall clock time.

Thirdly, potential interference between asynchronous data movement with the simulation's communications is minimized by properly scheduling data movement. The comparison of main loop time for the two different configurations shows that Staging may slow down the computation due to contention on the shared network, especially at large scales. However, by properly scheduling data movement, this interference is controlled to be less than 6%.

Overall, the reduction in visible I/O and operation times on compute nodes outweighs the interference experienced by the simulation due to asynchronous I/O and the insulating effects of decoupling the simulation I/O from variations in the file system performance improves the total execution time and reduces variation in the performance in spite of some increased latencies for performing some styles of operations. In terms of total CPU usage cost, calculated as total simulation time multiplied by total cores used, the Staging configuration is less costly when compared with the In-Compute-Node configuration at all scales (as shown in Figure 13(a)). There is a decline of savings from 8,192 to 16,384 cores mainly due to the interference of asynchronous data movement. At the scale of 16,384 compute cores, however, running

the simulation with the Staging configuration still saves 98 CPU hours in total compared with the In-Compute-Node configuration for a 30-minute simulation run. This suggests that the Staging approach helps GTC achieve better scalability in terms of total cost of both simulation and data preparation.

3.5.2.3 Offline Operations Discussion

Considerations for using offline operations are different from online operations. Instead of compute time and communication load being dominant factors, data storage requirements and file system interference generated are major concerns. Typically, offline operations, while slower to perform and much longer latency to completion, can be done cheaply or free. For operations that do not generate a reduction in data and instead generate approximately equivalent data in a different organization, such as sorting and layout re-organization, an offline approach would cost additional storage resource for intermediate data and meanwhile impact the file system by reading all of the data and writing it again. For example, when running at the scale of 65,536 cores, the particle data of GTC is 1TB per I/O dump. Offline sorting would cost 1 TB additional storage space every 120 seconds and the entire 1 TB would have to be read back in before it is rewritten. This moves the data through the disk controllers three times rather than once. Secondly, given the huge volume of GTC data, the read and write latency would be hundreds of seconds making the offline approach unsuitable for online data monitoring. For these sorts of operations, in-transit data manipulation is a big win.

For operations like the histogram and 2D histogram, the advantage of in-transit is still present. Using the same 1 TB per I/O dump output, two trips through the disk controller are required. While the output of this style of operation is comparatively very small, the impact of reading all of the data to generate the histograms both generates potentially large latency and long-term impact to the file system performance.

3.5.2.4 Evaluation of the DataSpaces Global Data Knowledge Service

To evaluate if the DataSpaces query engine can service queries on particle data in a timely manner without blocking the simulation between two successive I/O operations, a prototype implementation of the DataSpaces indexing and querying service is deployed on the staging nodes. The particles output from the GTC application are first sorted using the sorting operation, and then indexed by DataSpaces based on the *local id* and *rank* attributes to create a $2 \cdot 10^6 \times 256$ 2-D domain space. This spaces is then uniformly distributed across the DataSpaces compute cores in the Staging Area. On average, at all simulation scales ranging from 512 to 16,384 cores, the time required to fetch data from the GTC simulation is 20.3 seconds, sorting takes 30.6 seconds, and indexing takes 2.08 seconds. In total, it takes no more than 55 seconds for DataSpaces to prepare the data for query.

A test querying application that queries the entire domain space is deployed on additional compute cores (referred to as 'querying application cores' in subsequent text). In the experiments, the querying application cores partition the particle data among themselves and issue 11 consecutive queries to disjoint regions of the data. The particle sub-regions is 200MB in size for each querying application core. Since no a-priori knowledge is assumed about the existence of the particles data or its distribution, the first query includes query setup operations such as hashing, data discovery, query routing and data retrieval, and is significantly more expensive to perform as seen in Figure 14. However, it is a one-time cost and subsequent queries are much faster. The setup time shown in Figure 14 is an average value across the number of querying application cores and the hashing time is an average over the number of setup queries received at each core running a DataSpaces server in the staging area.

The query execution time for different numbers of querying application cores is plotted in Figure 14. The plotted times are an average over the number of queries

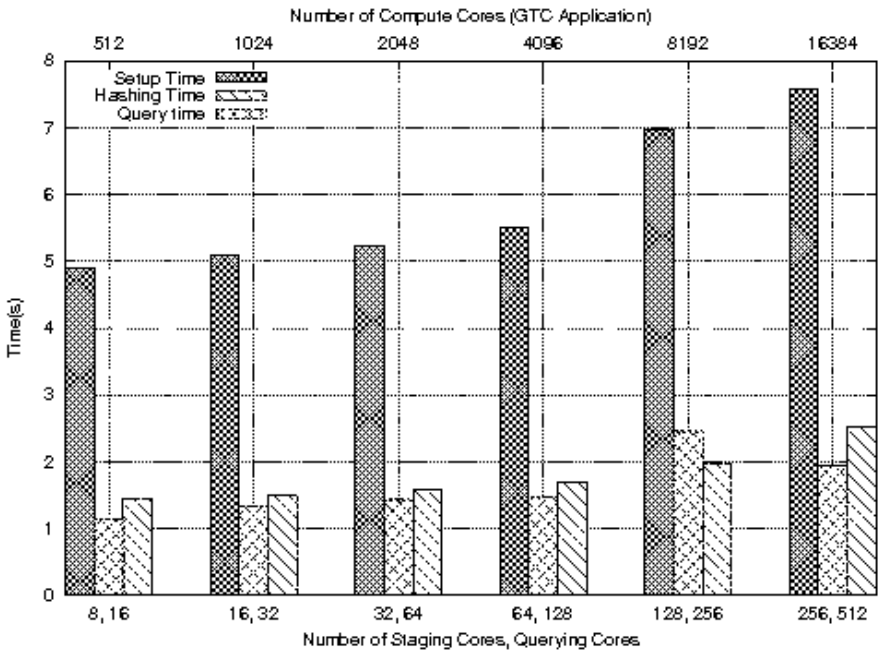


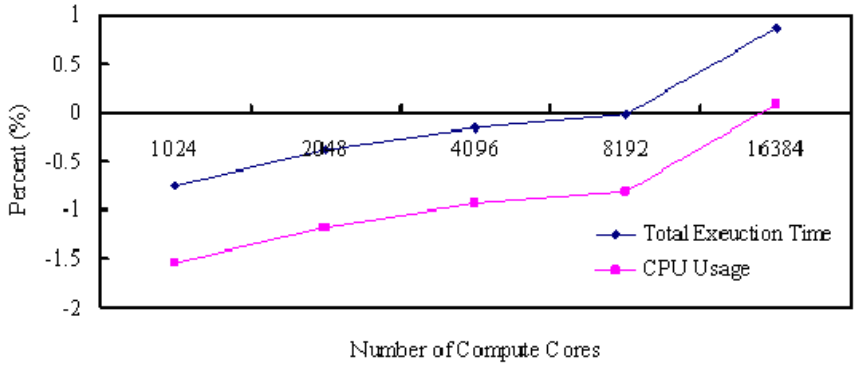
Figure 14: Setup, hashing and query time.

executed and over the querying application cores. The query time increases with the number of cores used since the domain size increases and is mapped to a larger number of cores in the staging area. In the presented example, one instance of the querying application receives replies to its query from multiple cores in the DataSpaces. The longer query time for the 256 application querying cores is due to load variability and interferences in the host system – we are investigating this further.

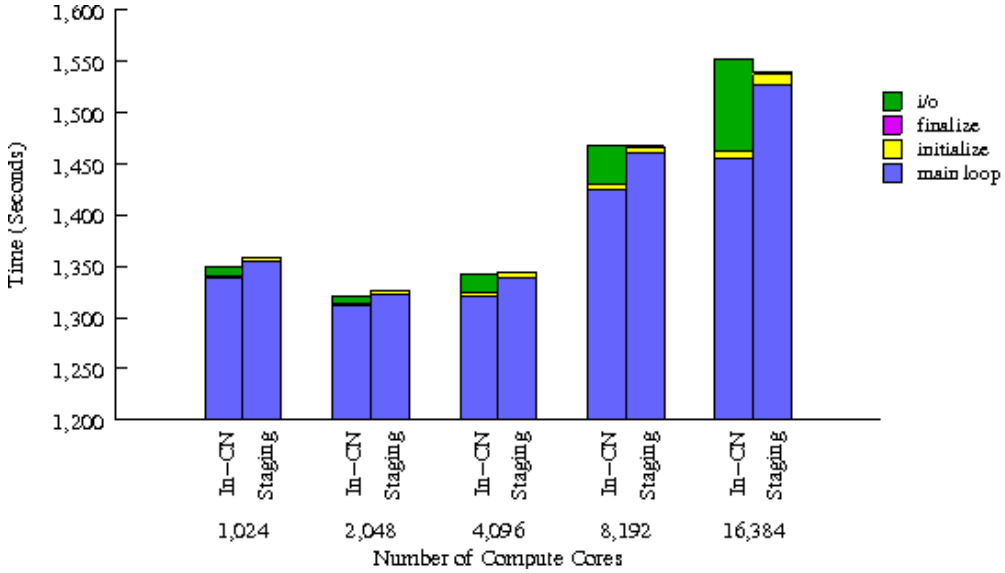
Note that DataSpaces indexes particles data and responds to all queries in less than 80 seconds. Considering the I/O interval is about 120 seconds, such an online query service can function effectively and without blocking the simulation.

3.5.3 Pixie3D Performance

Pixie3D performance is evaluated on the XT4 partition of Jaguar. Production runs use one MPI process per compute core. The data output from each process mainly consists of eight double-valued arrays. Each local array is part of a 3D global array, respectively. The tested setting uses a 32x32x32 local array size, which is a typical



(a) Improvement of Pixie3D simulation performance and cost



(b) Pixie3D total execution time breakdown

Figure 15: Pixie3D simulation performance.

setting for production runs. For each run, the simulation performs I/O about every 100 seconds. The ratio of compute cores to staging cores is maintained at 128:1 during weak scaling. Each process generates about 2 MB of data making this ratio workable.

Pixie3D is tested with an In-Compute-Node configuration and a Staging configuration. For the In-Compute-Node configuration, each MPI process writes output data to a single BP file using the ADIOS synchronous MPI-IO method. This results in a file in which local array chunks are scattered. In the Staging configuration, output data of compute nodes are sent to the Staging Area where they are merged to form

larger, contiguous global arrays.

Figure 15(b) shows the total simulation execution time for both the In-Compute-Node and Staging configurations. The Staging configuration slows the simulation in most cases by 0.01% to 0.7% when compared against the In-Compute-Node configuration. Unlike GTC, Pixie3D does not have enough computation intensity for asynchronous I/O to be an effective technique for offloading data. In each iteration, the inner loop of pixie3d performs collective communications (MPI_Reduce and MPI_Bcast) multiple times and between the mass communications are computations that only last about 0.7 seconds making it difficult to overlap data movement with computation without impacting the intensive messaging. The results show that the main loop time is increased because of asynchronous data movement. Although the I/O blocking time is well hidden, since it is such a tiny portion of the total execution time, this savings cannot outweigh the slowdown of computation due to communication interference. The operations tested for the GTC application were all intended to be performed before any data analysis were performed in order to speed read operations. The same is true for this data reorganization operation. While GTC's operations were a win-win for both writing and reading at all scales, Pixie3D's data reorganization requires larger job sizes to reach a tipping point where simulation performance can be improved by employing staging. Figure 15(a) shows the total cost of CPU seconds. As the simulation scales up, the I/O overhead weighs more in total execution time, and hence the impact of computation caused by data staging becomes less evident. Overall, there is a trend that the cost of Staging approach catches up with that of In-Compute-Node approach with increased simulation scale.

It is worth examining the savings generated during reading operations due to the reorganized data. Figure 16 shows the read performance on two files generated by two 4096-compute-core runs with Stingy and In-Compute-Node configuration, respectively. This result, along with the simulation cost shown in Figure 15(a), shows

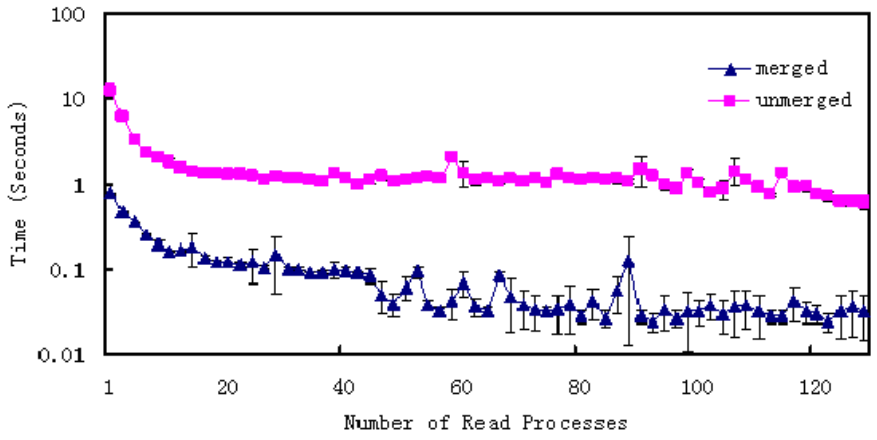


Figure 16: Time to read one global array of one time step from two 80GB BP files. 'merged' denotes the read time from a file written from Staging Area and 'unmerged' denotes the read time from a file written from compute nodes directly. Both files are generated by 4096-compute-core runs.

that at the scale of 4096 compute cores, 0.93% additional cost in simulation yields 10 times improvement in read performance of output data. This saving is more evident as scale increases.

In summary, the performance results show that in-transit data manipulation enabled by PreDatA middleware can improve the latency to operation completion compared with offline approach, reduce overall wall clock time of simulation even compared to online configuration at large scales, and reduce the impact on the shared file system when compared against both online and offline configurations. It is also shown that high-level data services can be efficiently built on top of PreDatA middleware.

3.6 Related Work

In this section we summarize previous research related to PreDatA work.

Scalable I/O and Data Analytics. Efficient access, understanding and management of voluminous and complex data generated by scientific simulations presents daunting challenges to both computational and computer scientists [53, 110]. Recent

work in parallel file systems [28, 107, 146] and I/O middleware [66, 88, 115, 154, 158] aims at optimizing data storage and access for scientific application workloads. Beyond pure high I/O bandwidth, however, scientists also require complex data analysis, search, and visualization technologies to facilitate better understanding of their data. Specialized data preparation, such as sorting, filtering, and indexing, is needed before data can be understood or visualized [29, 120, 132]. Our work extends the I/O middleware stack to exploit computational power along the output data flow to perform data preparation, characterization, and re-organization, which would facilitate subsequent data analysis.

Data Staging and Offloading in supercomputers. Previous work on data staging and asynchronous I/O [17, 42, 77, 78, 95, 103, 126] derives substantial performance advantages from hiding I/O latency with asynchronous data movement. Our recent work [11, 12] shows the importance of minimizing interference of asynchronous data movement with the application to achieve overall improvements in simulation time. One observation is that the computational resources on staging nodes are often under-utilized and the time intervals between I/O dumps are sufficiently large for extra processing on buffered data. In this chapter, we take one step forward and demonstrate the use of staging nodes for a diversity of data operations to achieve not only high write performance, but also high read performance and timely monitoring of output data and simulation.

Active Storage. Active Storage [112] deploys data processing operations directly on the storage nodes where the data are buffered to reduce the amount of data movement between storage and compute nodes. The storage nodes have limited computation and memory resources which are shared among applications, so one potential problem with Active Storage is how to manage such resources to meet deadlines for multiple applications and minimize performance downgrade of storage nodes. Abacus [18] demonstrates the benefit of flexible, dynamic function placement in Active Storage,

and we are going to investigate similar mechanisms for PreDatA.

In-situ Data Analytics and Visualization. Hercules [138] applies an end-to-end approach to tightly couple all simulation components, including meshing, partitioning, solver, and visualization, and runs all components on the same supercomputer. It eliminates intermediate I/O and data movement between simulation components to address the I/O bottleneck, but requires scaling data analysis and visualization to the level where simulation runs and all simulation components must be changed to efficiently share data with each other. PreDatA couples the Staging Area with the application more loosely and through the ADIOS interface, thereby requiring minimal changes to application code and providing improved flexibility in composing the simulation's output and analysis pipeline.

Scientific Workflows. Scientific workflow systems like Pegasus [40] and Kepler [93] are used to automate scientific data and simulation management. Unlike the end-to-end approach used in In-situ visualization mentioned above, components in the workflow are usually connected via a file-based interface, so that the performance of the workflow is very sensitive to data placement and movement and is easily affected by poor I/O performance [39]. PreDatA can serve as an early stage in output pipeline to apply application-specific data reduction, validation, and filtering operation before data is moved to disks, to reduce the data volumes to be processed in subsequent workflow steps.

Scientific Data Stream Processing. Scientific data stream processing, such as filtering [22], sampling [144], query [84], and transformation [70] complements our our work. This is because PreDatA can be used either as an in-transit data processing framework for implementing streaming processing tasks, or as a data forwarding layer to directly feed data to existing streaming processing systems.

Code Coupling. Memory-to-memory code coupling addresses some of the issues faced by PreDatA, such as data movement and re-distribution [13, 76]. PreDatA provides the underpinnings for supporting the rich model-model communications needed for inter-application interactions [41].

Interactive Computational Steering. Runtime steering can aid scientists in debugging and monitoring their simulations [54, 137]. The capability of extracting and inspecting data from running simulation with small overhead and interference makes PreDatA a potential infrastructure for online steering of running application.

Data-intensive Computing in the HPC Domain. Recently, there is increased interest in building high-level abstractions and programming models for data intensive applications in HPC domain. HiMach [136] applies the MapReduce model to analyze molecular dynamics simulation trajectories and shows some efficiency at tera-byte scale. In contrast, experiences from implementing materialized ground models [124] show poor performance of MapReduce because some of the features provided by MapReduce are unnecessary for its target application. AllPairs [99] gains similar insights in that a mismatch between the application workload and the available MapReduce abstractions can result in poor performance. The two-pass streaming model used by PreDatA appears sufficient for the applications we have used, but it remains an important item of future work to investigating higher level abstractions and a suitable programming model for future PreDatA codes.

3.7 Conclusions

This chapter presents the PreDatA middleware for preparing and characterizing data "online", that is, while data is being produced by the large scale simulations running on peta-scale machines. PreDatA offloads output data from a running simulation with low-overhead using asynchronous data extraction. It also exploits the computational power of staging nodes residing on the peta-scale machine and associated with each

large-scale application to perform select data manipulations. PreDatA enhances the scalability and flexibility of current I/O stacks on HEC platforms and is useful for data pre-processing, runtime data analysis and inspection. The DataSpaces services now being integrated into PreDatA also demonstrates its potential utility for rich model-model interactions in large-scale HPC codes. Performance evaluations with several production scientific applications on ORNL's peta-scale machines show the feasibility of the PreDatA approach as well as the performance advantages derived from using the PreDatA I/O stack compared to existing synchronous approaches.

CHAPTER IV

I/O MIDDLEWARE FOR LOCATION-FLEXIBLE ANALYTICS

In Chapter II, we use performance model to reveal that placement of online data analytics can significantly impact the end-to-end performance and that there is a consequent need for flexibility in the location of data analytics. In this chapter, we propose an I/O middleware called FlexIO which enables flexible analytics placement.

4.1 Introduction

4.1.1 Need for Location-Flexible Data Analytics

For real-time processing of the outputs generated by large scale simulations, a key problem to address is "where" analytics are placed along the I/O path: on compute nodes integrated with application codes, on compute nodes as separate software components, on nodes dedicated to analytics (also termed "staging nodes"), or offline (after data is placed into persistent storage) (as illustrated in Figure 1 in Chapter I). Placing data analytics involves deciding the resources to allocate to analytics computation and realizing the data movements between simulation and analytics. Experimental results and analytical models in pervious chapters show that analytics placement can significantly impact the performance (e.g., runtime) and cost (e.g., CPU hours) of the coupled simulation and analytics and that the best placement depends on the particular analytics codes, data volumes, scale of operation, and machine characteristics. The consequent insight is that no single, specific placement will be "best" for all applications and analytics.

Such variation has important implications to both scientists and the software that

supports analytics. Scientists desire the performance benefit from good placement, but it is a burden for them to tune placement every time a different analytics code is run, especially when this requires significant coding effort. There is a need, therefore, for infrastructure that makes it easy to decide, enforce, change, and tune analytics placement. At the same time, if such an analytics software infrastructure aims to support a broad range of simulations and analytics, lacking placement flexibility limits its applicability, since fixed placements may cause negative or even disastrous impact on application performance at large scale. Flexible placement, therefore, is a critical element of analytics infrastructure.

Most existing systems support certain, fixed placement choices and therefore, each is efficient or applicable to certain classes of analytics. Some permit analytics to run at different locations, but require adopting particular coding patterns or re-placement involves substantial re-coding. Further, they do not support seamlessly switching analytics between online and offline, nor do they allow dynamic changes in analytics placement. Typical causes of limited flexibility are (1) an inability to handle the alternative data movements between simulation and analytics required by different placement options (i.e., supporting inter-node and intra-node data transfer and file I/O); (2) lack of uniform higher-level interfaces that hide data movement detail; (3) imposition of specific computation models for analytics; and (4) inability to achieve those requirements with high performance and scalability.

4.1.2 Technical Contributions

The FlexIO middleware described in this chapter is designed to provide data movement between simulation and analytics with both high performance and location flexibility. It offers the following functionalities:

1) Flexibility in where analytics codes are run – on compute nodes, on staging

nodes, and/or any combination thereof. This is realized through FlexIO's high performance intra- and inter-node data movement transports which are implemented with shared memory queues and RDMA, respectively, and tuned for high throughput, contention-avoidance, and memory efficiency.

2) Analytics placements can be altered without requiring application codes to be changed. FlexIO's high level programming interface makes changes in placement transparent to simulation and analytics codes. Users can even seamlessly switch analytics to run offline when there are insufficient online resources for their timely execution.

3) Runtime performance monitoring collects information about computation and data movement that is useful to scientists and automated runtime management systems for performance understanding and placement decisions.

4) Mobile codelets, termed "Data Conditioning (DC) Plug-ins", can be dynamically deployed and migrated along the I/O path, to perform useful on-the-fly data manipulation such as data selection, sampling and transformation.

5) FlexIO enables various placement policies to exploit the location flexibility for tuning application performance, CPU usage, and data movement cost. Based on FlexIO, we implement a holistic placement policy which reduce both inter and intra program data movement costs. We also devise a node topology aware policy which takes into account the impact of cache topology on analytics placement.

FlexIO operates on both Infiniband and the new Cray XK6 with Gemini interconnect. FlexIO has been used to implement in situ analytics for two leadership scientific applications: GTS fusion simulation and S3D combustion simulation. Experiments show that leveraging the flexibility enabled by FlexIO to tune placement can improve total execution time by up to 30% compared to inline-only solutions and the benefit is more evident at larger scales.

The remainder of this chapter is organized as follows. Section 4.2 describes the

design and implementation of the FlexIO middleware. Section 4.3 describes how to automate placement of analytics driven by performance and cost metrics. Section 4.4 shows performance improvements for two large-scale scientific applications due to flexible analytics placement. Section 4.5 reviews related work and Section 4.6 concludes this chapter.

4.2 FlexIO Design and Implementation

4.2.1 Overview

The FlexIO software stack is depicted in Figure 17. Simulation and analytics codes use the ADIOS [88] read/write API for data exchange. The FlexIO runtime handles buffer management, parallel data re-distribution, and performance monitoring. It also manages Data Conditioning Plug-Ins which are mobile codelets compiled, deployed, and executed at runtime for on-the-fly data manipulation. Runtime performance monitoring provides information for scheduling data movements and for dynamic DC Plug-in placement and can also be retained for offline performance tuning. At the lowest transport level, FlexIO uses efficient RDMA and shared memory data movements for inter- and intra- node movements, respectively. The choice of low level transport is automatically configured according to the placement of online analytics.

FlexIO leverages the ADIOS [88] parallel I/O library, which provides meta-data rich read/write interfaces to simulation and analysis codes. ADIOS has a set of built-in I/O methods under its higher level API to support various file I/O methods (such as MPI-IO, HDF5, and NetCDF) as well as data staging methods [12]. Switching between different methods can be configured through an external XML configuration file, without modification to application codes. ADIOS has been used by several leadership scientific codes and is integrated with popular analysis and visualization tools that include Matlab, ParaView, and VisIt. FlexIO inherits from ADIOS useful features like its high level API and file I/O methods (to enable offline placement), and

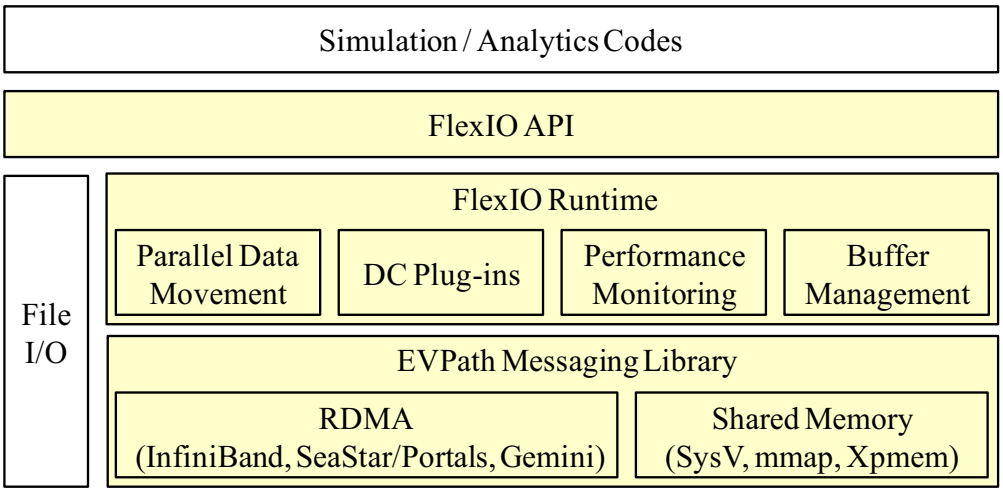


Figure 17: FlexIO software stack.

its implementation benefits from our previous work on RDMA-based data movement and staging. New functionalities compared to such prior work include an ADIOS streaming read interface, methods for parallel memory-to-memory multi-dimensional array re-distribution, efficient data movement for Cray XK6 systems with the Gemini interconnect, cache efficient on-node data movement via shared memory, intelligent buffer management, enriched performance monitoring, and mobile DC Plug-ins. In total, FlexIO is a system that supports diverse analytics placements via built-in efficient methods for data movement between a simulation and analytics components.

4.2.2 High Level Interface

The high-level interface of FlexIO extends the existing ADIOS file read/write API with three goals: 1) expressiveness in supporting common I/O patterns for simulation and analytics codes; 2) backwards compatibility with the existing ADIOS file I/O interface; and 3) easily switched underlying transports.

Conceptually, the FlexIO interface allows simulations to pass data to analytics via "files", and to operate on these "files" in either file or stream modes. In both modes, the data model is compatible with the existing ADIOS data model, where the simulation output data is logically time-indexed, and each timestep of output data is

a group of variables of scalar or array types. In the file mode, data is written to the file system and read back by analytics, using one of ADIOS' file I/O methods. The file mode is for backwards compatibility with the existing ADIOS file I/O interface.

The newly added stream mode is specifically intended for memory-to-memory data movement between simulation and online analytics. Here, the simulation creates a "file" with some unique name, and the analytics opens the named "file", but internally, this establishes connections to simulation processes via the underlying transport. Simulation processes, then, periodically write data to the "file", and the data is passed to analytics as return parameters of their read calls (again, the underlying transport handles actual data movement). When the simulation closes the "file", the connections are closed by the transport and analytics components receive End-of-Stream as return values from their read calls. As a result, stream mode is compatible with file I/O in that it can be switched with file mode without code changes.

For stream mode, there are two common I/O patterns for high end applications. The first is for process-group-oriented data exchanges: during each I/O timestep, the variables written from each simulation process are conceptually packed into a group, called "Process Group", and the analytics specifies the process groups it wants to read by simulation processes' MPI ranks. The other pattern is a global array data exchange, where some multi-dimensional array, distributed among several simulation processes, is passed to several analytics processes. As in other MxN data exchanges [13], however, analytics processes may specify an array distribution or layout different from that present on the simulation side. In response, FlexIO properly chunks, splits, transfers, and re-organizes the array data exchanged between simulation and analytics, as shown in Figure 18.

The high-level API makes it easy to change underlying transports, without the need to change applications. A one-line update to the configuration file is sufficient

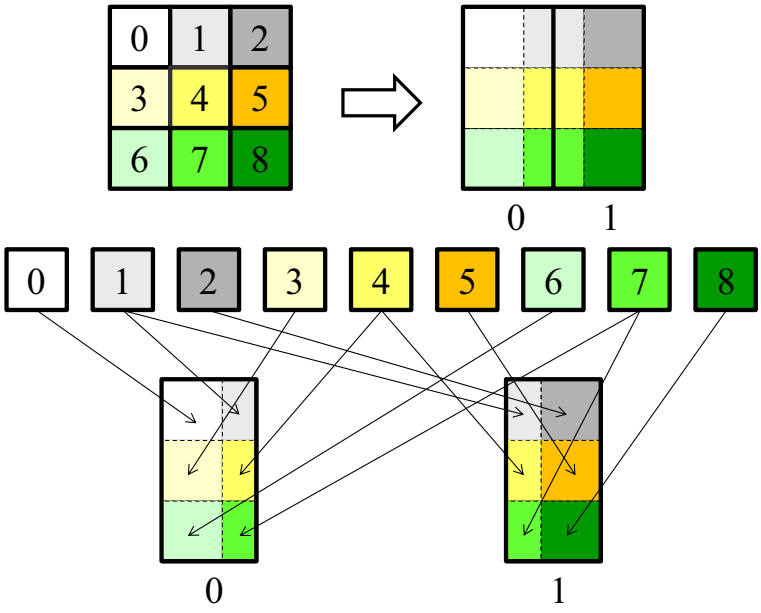


Figure 18: Global array re-distribution. A 2D array is distributed among 9 simulation processes and passed to 2 analytics processes.

to switch between file I/O and online data movement transports, and intra- vs. inter-node transports are automatically configured according to the placements of communicating simulation and online analytics processes. To tune transports, transport-specific parameters specified as hints in an XML configuration file are passed to the FlexIO runtime. We refer readers to [14] for details of the interface specification.

4.2.3 Data Movement Protocols

There is considerable complexity in presenting to end users a convenient API, yet also providing placement flexibility and high performance. Key to this complexity is that the FlexIO runtime must translate the high-level API calls into actual data movements between simulation and analytics processes using low-level RDMA or shared memory transports. As shown in Figure 18, the MxN mapping, i.e., which simulation process should send which piece of its data to which analytics processes, is determined by the overlapping portion(s) of data specified in the simulation's write and analytics' read calls, respectively. Establishing the necessary connections and transferring data efficiently at large scales is a non-trivial task. Below we describe

the connection management and data transfer protocols used by the FlexIO runtime.

1) Connection Management.

Before actual data movement, simulation and analytics services connect to each other via assistance from an external directory server. To avoid overloading this server, simulation and analytics processes, respectively, elect a local coordinator. When creating a file in stream mode, the coordinator of the simulation registers with the directory server a file name associated with its own contact information. When the analytics opens that file, its coordinator looks up the server with the file name, retrieves the contact information of the simulation's coordinator, and makes a connection with it. The directory server is involved only in discovery and connection setup and is not in the critical path of actual data movements.

2) Data Movement.

To move global array data between two parallel programs, array data must be transferred according to the data distributions at both sides. The FlexIO write and read API captures the array distribution among simulation and analytics processes, respectively. Based on this information, the FlexIO runtime generates the re-distribution mapping. At each side, coordinators first gather array distributions for all processes (Steps 1.s and 1.a, respectively), exchange the distribution information with each other (Step 2), and then broadcast the peer-side distribution to all processes (Step 3). At this point, each process knows the array distribution of all other peer processes, so that it can calculate the mapping independently. Each sender process packs strides for each receiver process with overlapping array index range, and sends the packed strides to each receiver process (Step 4.s). Each receiver prepares a receive buffer based on the mapping and copies received strides into the appropriate target buffer (Step 4.a). The Process-Group-oriented data movement pattern is implemented in a similar fashion.

There are several optional optimizations. First, write side calls can be either

synchronous or asynchronous. The asynchronous API helps overlap data movement with other activities like the simulation's computation. The second optimization is batching. The default granularity of data movement is per-variable. Users can instruct FlexIO to pack multiple variables and transfer them in a batch. This will cause both handshaking and data messages to be aggregated.

The third optimization is caching to reduce the cost of handshaking. By default, the complete handshaking protocol (Step 1 to 4 as described above) is performed for each variable at each I/O timestep. If distribution information and buffer addresses are unchanged across timesteps, then some or all of the handshaking steps can be avoided by reusing existing information from previous timestep. The sender or receiver can inform the FlexIO runtime about three possible caching options:

i) NO_CACHING: perform the full handshaking protocol;

ii) CACHING_LOCAL: re-use local side distribution information (skip Steps 1), but still exchange distribution information with peer side (perform Step 2 to 4);

iii) CACHING_ALL: re-use both local and peer sides' distribution data, so that handshaking is completely avoided.

FlexIO uses the EVPath messaging library [45] to implement its data movement protocols. EVPath provides point-to-point messaging and data marshaling capabilities. Its modular architecture supports multiple messaging transports, and we have added to it the shared memory transport and the RDMA transport required by FlexIO.

4.2.4 Shared Memory Transport

The shared memory transport is for intra-node data movement. Using it, small messages like handshaking messages are passed through data queues in shared memory segments. Each data queue is a single-producer, single-consumer, circular, lock-free FIFO queue inspired by Fastforward [52]. The producer and consumer have separate

pointers to the next entry to enqueue or dequeue, and these pointers are guaranteed to be placed into different cache lines to reduce cache coherency traffic. Each entry in queue has a payload field of fixed-size and a status flag with two possible states: full or empty. Entries in data queues are carefully aligned and padded to make sure they do not share cache lines, so as to reduce false sharing. During data movement, the consumer polls the flag of the next entry to dequeue. The producer first checks that the next entry to enqueue is marked as "empty" before copying data into it. The flag is then set to "full"; this signals the consumer, which then copies data from the entry into the target receive buffer and sets flag to "empty" to release the entry to the producer. On systems with weak memory consistency, additional memory fences are inserted.

For large messages such as actual simulation output data, a shared memory buffer pool is used. The producer pre-allocates a shared memory buffer pool indexed with a free list. When sending a large message, the producer tries to find a buffer of the closest size in the pool (and allocates one if not found), copies the message into it, sends a control message to the data queue, and returns if it is an asynchronous movement. The consumer extracts the address and length from the control message, copies data from the shared memory buffer into target buffer, and returns the buffer to the producer's free list. Thus two memory copies are needed for sending large messages asynchronously.

On the Cray XK platform, our shared memory transport leverages page mapping support from the XPMEM kernel module [6] to reduce memory copy overheads. During synchronous large message transfers, the producer makes its source buffer available for sharing by calling xpmem_make(), and sends the shared memory segment id through the data queue. The consumer then gets the memory handle, maps the producer's send buffer into its address space, and copies data to the target receive buffer.

4.2.5 RDMA Transport

The RDMA transport in EVPath is for inter-node data movement. It is built on top of Sandia National Laboratory's NNTI library [62]. NNTI implements a uniform set of APIs (including Connect, Memory Register/Unregister, RDMA Put and Get) above ibverbs, Portals, and uGNI. It therefore, provides a portability layer among different interconnects (IB, SeaStar and Gemini). Based on NNTI, the EVPath RDMA transport implements buffer management and several optimizations for high performance RDMA data movement.

Dynamic buffer allocation and memory registration can cause significant overheads in RDMA-based data movement. Figure 19 demonstrates this with a point-to-point RDMA Get bandwidth test on the Cray XK 6. This is particularly an issue for applications generating particle data, since the number of particles written by a simulation process may change each timestep due to particle movement. One solution to reduce this cost is to use a persistent buffer and registration cache, as in MPI [116] and Charm++ [133]. We use a similar approach: allocated and registered send and receive buffers are temporarily kept in a buffer pool; later data transfers try to reuse those buffers whenever possible. A configurable threshold value controls total memory usage and triggers buffer reclamation, if necessary.

For small messages, a pair of message queues is established between two interacting processes for two way messaging. The sender process uses NNTI's RDMA Put to send a message into the receiver process' message queue. For the Cray Gemini interconnect, this uses FMA Put to send the data. For large message transfers, we use receiver-directed RDMA Get for data movement. The sender process first copies the message into a send buffer acquired from the buffer pool and sends to the receiver a small control message containing the address and size of the send buffer. The receiver prepares a receive buffer, and issues RDMA Get to fetch data according to some scheduling policy. For Gemini, RDMA Get is implemented with uGNI's BTE RDMA

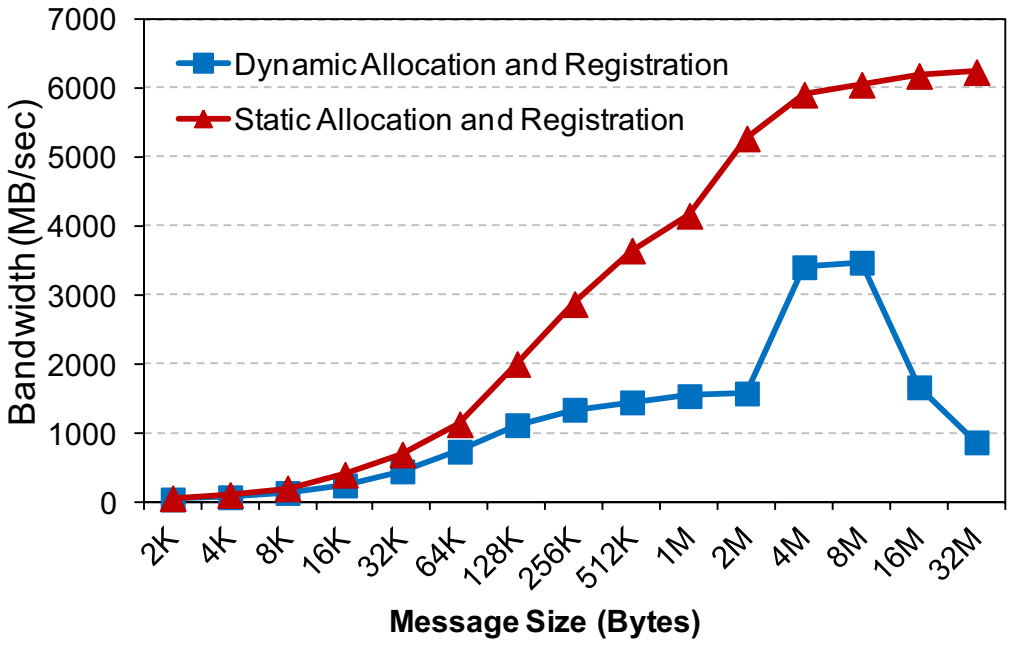


Figure 19: Cost of dynamic buffer allocation and registration in RDMA Get on Cray XK6 with Gemini interconnet.

operation. The scheduling technique is leveraged from our previous work in data staging [12]; and its use can effectively reduce network contention.

4.2.6 Data Conditioning Plug-ins

Data Conditioning Plug-ins are mobile codes embedded in the FlexIO transport. They are triggered to perform operations on data during the exchange of data between simulation and analytics. DC Plug-ins can be executed within the address space of either the simulation or analytics, and they can be migrated across address spaces at runtime.

DC Plug-ins are stateless codelets created on the reader side (e.g., analytics) to customize writer-side outputs on the fly. Useful examples of DC Plug-ins include data markup, annotation, sampling, bounding box, unit conversion, etc. They are typically lightweight in terms of compute and memory usage, and are easily programmed with the C subset offered by the C-on-Demand (CoD) [47].

DC Plug-ins are specified as parameters to FlexIO read API calls. Their code

strings are compiled and installed in the appropriate process' address space through the dynamic binary code generation offered by CoD. The code can be executed at either the analytics side or simulation side. Runtime deployment of DC Plug-ins from the analytics side into simulation processes is through a communication channel separate from the ones used for data movement. DC Plug-in placement is informed from the caller. Compared to our previous work [10], DC Plug-in has better scalability and is fully integrated with the FlexIO infrastructure; we have also implemented various runtime data manipulation functionality and management policies with DC Plug-ins to further enhance the I/O path (more details in Section 4.4).

4.2.7 Performance Monitoring

FlexIO monitors the performance of simulation, analytics, and DC Plug-ins. There are measurement points at all levels of the FlexIO software stack to gather a variety of information, including the timing of data movement and DC Plug-in execution, as well as transferred data volumes. Dynamic memory allocation points within FlexIO are also instrumented to record memory usage during data movement. Optionally, information about the computation and communication behavior of simulation and analytics can also be obtained by explicitly instrumenting the codes.

Performance information is used in two ways. For offline performance tuning, monitoring information can be dumped to trace files, and the developer can use it to understand and tune analytics codes. For runtime management, monitoring data captured from the simulation side can be gathered online and transferred to the analytics side. The analytics process(es) can then use it to dynamically schedule data movement and decide the placement of DC Plug-ins.

4.3 Exploiting Placement Flexibility

FlexIO makes it possible to tune analytics placement to improve performance and/or reduce cost: it provides performance information that can aid in placement decision,

and it can automatically configure the underlying transport to enforce any placement decision made by users. To illustrate the importance of this, this section describes three placement algorithms realizing different placement policies. Our purpose is not to show "best" policies, but instead, to show how FlexIO makes it easy to implement alternative methods suitable for different usage scenarios. This is important because there may be frequent changes to analytics codes and to the configurations of I/O pipelines, to support evolving scientific processes. The heuristic algorithms shown find satisfactory placements for large scale simulation and analytics within reasonable time frames. They assume that simulation and analytics exhibit steady runtime behavior so that placements can be statically determined and enforced at job launch time. Such assumption holds for most of the practical use cases we have encountered.

Placement algorithms: 1) optimize some objective (e.g., minimizing total execution time); 2) use a resource allocation policy that determines how much resource to allocate to simulation and analytics components; and 3) carry out a resource binding policy that decides the process/thread to physical resource mapping.

4.3.1 Performance and Cost Objectives

The following performance and cost metrics are of interest to science end users.

Total Execution Time: the time from the start of simulation and analytics to the completion of both.

Total CPU Hours: the total nodes used multiplied by the total execution time (in units of hours). This metric measures the cost of a run, as supercomputing centers commonly charge users with the CPU hours consumed by their jobs.

Data Movement Volume: the amount of data moved between simulation and staged analytics.

4.3.2 Placement Algorithms

1) Data aware mapping. The data aware mapping algorithm introduced in [156] takes as input a communication matrix recording the data movement volume between simulation processes and analytics processes. It applies graph partitioning to divide simulation and analytics processes into as many groups as the number of nodes, and then assigns each process group to a node with each process mapped to one core. Data aware mapping is essentially a resource binding algorithm, and it tends to place frequently communicating processes from different programs onto the same node.

2) Holistic placement. We extend data aware placement to holistically treat two additional issues: i) to carry out resource allocation, in addition to simply deciding resource bindings, and ii) to also consider the data movements within parallel simulation and analytics programs (e.g., their MPI communications). Termed "holistic placement", we have experimented with two algorithm variants, for synchronous vs. asynchronous data movement scenarios, respectively. These algorithms take as input the input configuration of the simulation and the strong scaling function of analytics. Performance profiling is used to obtain such information.

When data movement between simulation and analytics is synchronous, holistic placement works as follows. During resource allocation, the analytics are scaled to match the data generation rate of the simulation. The idea is that simulation and analytics form a two-stage pipeline and hence, matching the analytics' data consumption rate with simulation's data generation rate leads to minimal pipeline stalls. The output of the resource allocation step is the number of processes needed to run analytics. During resource binding, the algorithm constructs a communication matrix that records both inter- and intra-program data movement. It models the target parallel machine as a two-level tree in which cores of the same node are siblings and have less communication cost with each other than with cores on different nodes. It then uses the graph mapping algorithm provided by the SCOTCH library [5] to map

the communication graph to the architecture graph. Compared to data aware mapping, holistic placement 1) captures the trade-off between inter- and intra- program communication, and 2) can be easily extended to model the machine architecture in greater detail (as will be shown in the third algorithm).

When data is moved asynchronously between simulation and analytics, the algorithm additionally considers the asynchrony effect of data movement. Asynchronous data movement overlaps with other activities such as the simulation's computation. Accordingly, unlike the synchronous case, the resource allocation step must ensure that the sum of data movement time and analytics computation time is no larger than the simulation's I/O interval. Data movement time is estimated as total data size divided by point-to-point RDMA transport bandwidth. This estimation is conservative because it assumes data are moved to analytics sequentially (from one simulation process at a time) through the interconnect instead of shared memory, and it may lead to resource over-provisioning for analytics. However, given that analytics usually runs at a much smaller scale than the simulation, such over-provisioning is unlikely to cost significant additional resources and may even be beneficial to accommodate variations in analytics running times. The resource binding step for asynchronous case is the same as for synchronous case described above.

3) Node topology awareness. To demonstrate the ease with which placement policies can be changed in FlexIO, we explore one additional generalization of the holistic mapping algorithm, designed to take into account the complicated cache topologies and deep memory hierarchies of modern multi-core processors. Figure 20 shows the memory structure of a machine with four quad-core AMD Barcelona processors and four NUMA domains. Cores share different levels of cache and memory resources, which results in non-uniform on-node communication times between cores.

Node topology aware placement, then, further extends holistic placement by modeling the target machine as a multi-level hierarchy: cores within the same node are

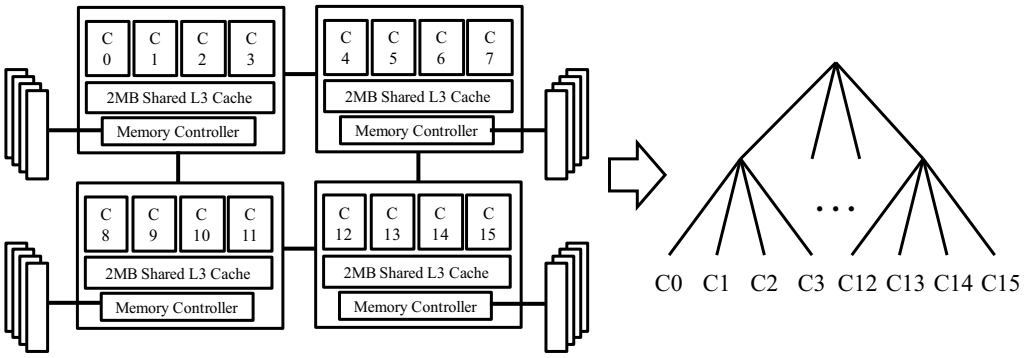


Figure 20: A multi-socket NUMA node architecture.

placed at different levels of the tree according to the cache topology. The graph mapping algorithm (used in Holistic Placement as described above) is then applied to map the communication graph onto the hierarchical architecture tree and generate process-to-core binding.

For NUMA machines, the algorithm not only decides process-to-core binding, but also determines the placement of FlexIO's internal buffers in memory. Our default policy is that the shared memory data queues and buffer pools are placed into simulation processes' local NUMA domain no matter where analytics processes are located. This arrangement facilitates the simulation's access to those data structures but may penalize analytics' access. The idea is that in most cases, the simulation is the performance-bounding part in the producer-consumer pipeline, while the analytics are more tolerant of slower data movement.

4.4 Performance Evaluation

In this section, we present experimental results obtained from tuning placements of analytics for two large-scale applications: GTS and S3D. We also demonstrate the utility of Data Conditioning Plug-ins to enable dynamic placement of analytics at runtime.

Experiments are run on Oak Ridge National Laboratory's Titan Cray XK6 and

Smoky cluster. Titan is upgraded from the Jaguar Cray XT5 and equipped with 18,688 compute nodes, 960 of which contain GPUs. Each compute node has a 16-core 2.2GHz AMD Opteron 6274 (Interlagos) processor and 32GB of RAM. The machine uses the Gemini interconnect. Smoky is an 80 node cluster. Each compute node has four quad-core 2.0GHz AMD Opteron processors (as shown in Figure 20) and 32 GB of memory. The Smoky cluster uses DDR InfiniBand interconnect. Both Titan and Smoky have access to the center-wide Lustre file system.

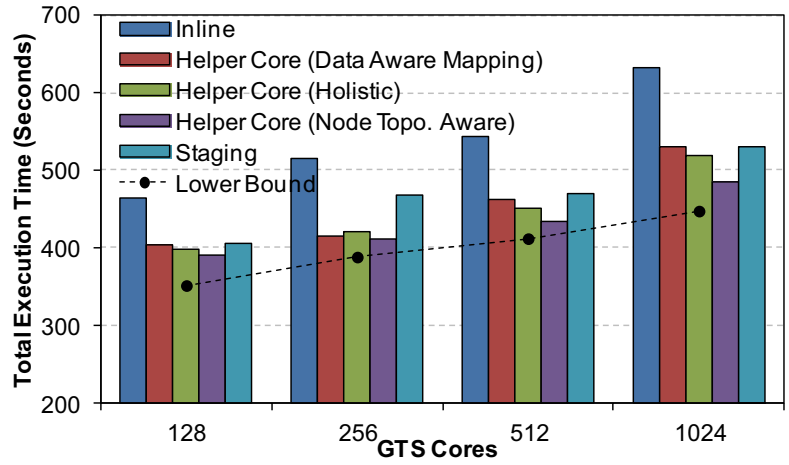
4.4.1 GTS Performance

GTS (Gyrokinetic Tokamak Simulation) is a global three-dimensional Particle-In-Cell (PIC) code used to study the microturbulence and associated transport in magnetically confined fusion plasma of tokamak torodial devices [139]. GTS simulation outputs particle data containing two 2-dimensional particle arrays for zions and electrons, respectively. The two arrays contain seven attributes for each particle, including coordinates, velocity, weight and particle ID. The particle data is processed by a series of analysis steps, including the calculation of particle distribution function and a range query on the velocity attributes of all particles. The query result is 20% of the original output particles. 1D and 2D histograms are generated from the query results and written to files which can then be used for parallel coordinates visualization. The analytics code uses FlexIO's streaming mode to read particles data and follows the process-group-oriented I/O pattern.

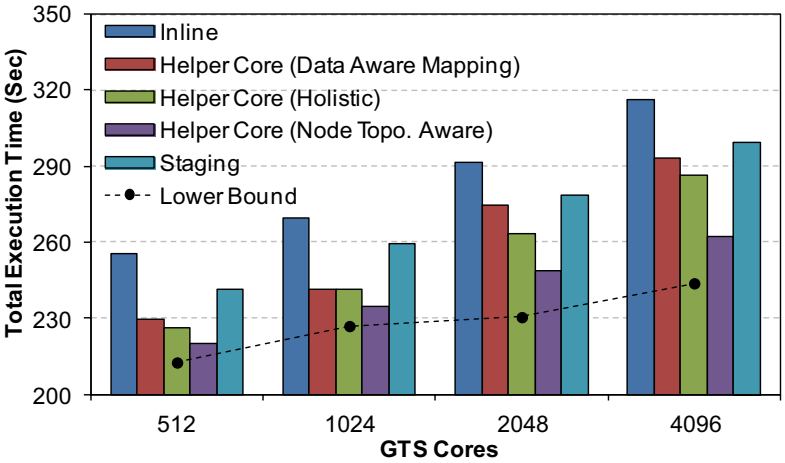
We run GTS with a typical production run configuration, which results in particle data output size of 110MB per process. GTS is run in OpenMP/MPI hybrid mode, as suggested by the GTS team. It outputs particle data every two simulation cycles, as desired by scientists.

1) Tuning Placement of Analytics.

We use the approaches described in Section 4.3 to place analytics for GTS. For



(a) Total Execution Time on Smoky



(b) Total Execution Time on Titan

Figure 21: GTS performance tuning on Smoky and Titan.

resource allocation, we apply the holistic policy to decide the number of processes to run analytics so that the data consumption rate matches GTS simulation's I/O frequency. After completing resource allocation, all three placement algorithms leverage inter-process communication volumes to determine process to core binding. Furthermore, since GTS itself can be strong-scaled for a fixed input problem size by varying number of OpenMP threads per MPI process, we decide the placement for each of GTS configurations with different number of OpenMP threads. We compare the resulting performance and cost of different configurations and placements.

Figure 21(a) shows the Total Execution Time of the coupled GTS simulation

and analytics with different placements at various scales on Smoky (weak scaling is applied). At all scales, all three algorithms decide to place analytics on Helper Cores in compute nodes (there are still differences among them, as will be explained later). The particular helper core placement found by node topology aware algorithm consistently shows the best performance: GTS is configured to run with 3 OpenMP threads per MPI process, and every 4 MPI processes are placed on each compute node; 4 analytics processes are placed on the remaining 4 cores of each node (i.e., the Helper Cores); GTS processes pass data to analytics processes through shared memory transport whose internal buffers are pinned in local NUMA domains. Besides, the GTS processes are placed onto nodes so that their 2D grid communication pattern is aligned with the target machine modeled as a 3-level tree.

In comparison, the holistic placement algorithm maps GTS and analytics processes onto nodes in the same way as node topology aware placement. However, since it ignores the NUMA structure of each node, it maps GTS threads and analytics processes linearly to cores within each node. OpenMP threads of some GTS processes are placed across NUMA boundaries, which hurts performance by up to 7.0% on Smoky. The placement found by data aware mapping algorithm has comparable performance as holistic placement. Largely this is because GTS performance is in-sensitive to process placement and hence ignoring its internal communication in placement decision does not cause notable performance penalty. However, data aware mapping is still outperformed by node topology aware placement by up to 9.5% due to its ignorance of NUMA structure.

We also place analytics inline and on a set of separate staging nodes. With inline placement (Case 2 in Figure 22), the GTS processes directly call analytics routine. On Smoky whose compute nodes has 16 cores each, we run GTS with 4 OpenMP threads per MPI process and place 4 MPI processes on each compute node. In comparison, the helper core placement takes 1 core from GTS and offloads analytics onto that

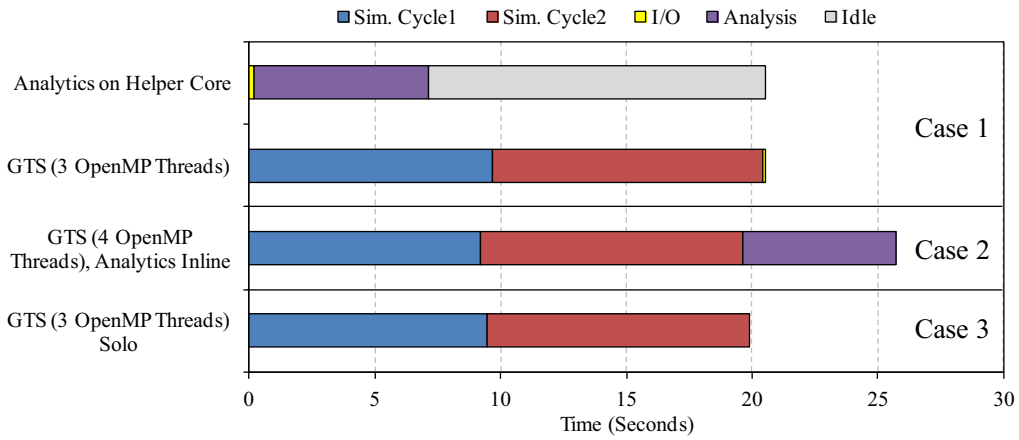


Figure 22: Detailed timing of GTS and analytics. GTS runs with 128 MPI processes on Smoky.

core (Case 1 in Figure 22). Such offloading is beneficial for two reasons. On one hand, GTS running with 4 OpenMP threads cannot make full use of all cores within a compute node due to the fact that there are code regions in GTS where only main thread is active. Taking 1 core out of 4 from a GTS process only slows down GTS by 2.7% (as indicated by the increase of simulation "cycle1" and "cycle2" time from Case2 to Case 1). On the other hand, the inline analytics weighs 23.6% of GTS runtime, so offloading analytics to helper core reduces Total Execution Time.

When placing analytics onto separate staging nodes, data are moved to staging nodes through RDMA transport. Compared to the helper core placement, the pitfalls of placing analytics in staging nodes are: 1) huge amounts of particle data are moved through interconnect which consumes more power than on-node movement; 2) asynchronous bulk data movement can interfere with simulation's MPI communication. We have to carefully set the asynchronous data movement scheduling policy to keep the GTS slowdown under 15%.

In terms of CPU hours cost, Inline placement is the worst due to penalty of running non-scalable analytics at large scales. Helper core placement use the same core counts as Inline placement but consumes less CPU hours by finishing faster. Staging placement is worse than helper core placement since it allocates additional

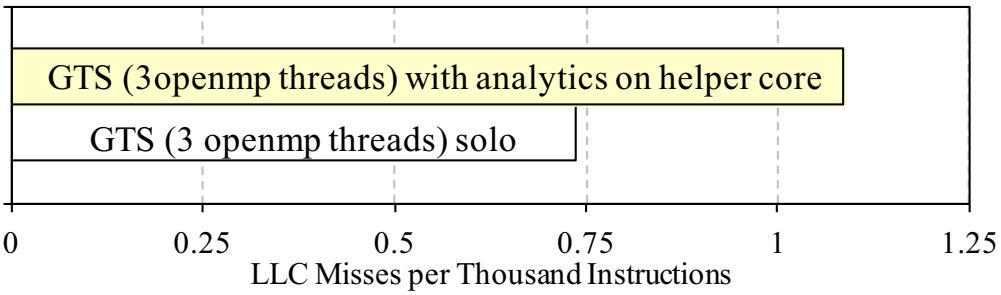


Figure 23: Last level cache miss rates of GTS on Smoky.

nodes but does not achieve better total execution time.

In terms of data movement volume, both inline and helper core placement avoid moving particle data between simulation and analytics through interconnect, while staging placement causes all particle data moved through interconnect. On the other hand, since staging placement maps analytics processes closer to each other than the other two placements, staging placement helps reduce the amount of analytics' internal MPI communication which go through interconnect. Overall, since inter-program data movement is dominant and analytics runs local query to reduce data, helper core and inline placement reduces inter-node data movement by about 90% over staging placement.

Figure 21(b) shows placement tuning results on Titan. Similar to Smoky results, on Titan which has 2 NUMA domains and 8 cores in each, running GTS with 7 OpenMP threads per MPI process and analytics on a separate helper core within the NUMA domain results in the best performance and cost.

2) A Closer Look at Helper Core Placement.

Figure 22 (Case 1) shows that GTS and analytics experience nearly invisible I/O overhead thanks to the shared memory transport. It also shows that analytics processes are idle for 67% of time, indicating over-provisioning for analytics due to our conservative resource allocation policy.

The downside of placing simulation and analytics on the same node is interference between them due to contention on shared on-node resources. To assess such interference, we test two cases: i) GTS with 3 OpenMP threads runs in solo and does no I/O or analytics (Case 3 in Figure 22) vs. ii) GTS with 3 OpenMP threads runs with analytics placed on helper cores (Case 1 in Figure 22). Figure 23 shows the aggregated L3 cache miss rate (measured in L3 cache misses per 1K instructions) seen by all GTS threads in simulation main loop in two cases (hardware performance counters are recorded with PAPI [4]). GTS experiences 47% more L3 cache misses when analytics runs on helper core and share L3 cache with it, and its simulation time ("cycle1" and "cycle2" in Figure 23) increases by 4.1%. Achieving better performance isolation between simulation and analysis when they are placed on the same node is part of our future work.

3) How Close is Our Solution to the Optimal?

The runtime of GTS which runs solo with 4 OpenMP threads and does not perform I/O or analytics can be considered as the Total Execution Time when data movement and analytics are "free" (no resource usage) and infinitely fast. This value is therefore less or equal to the optimal Total Execution Time of coupled simulation and analytics. The best placement solution which we have found is at most 7.9% larger than this lower bound (dashed lines in Figure 6) with the same core count used at all scales on Titan, and at most 8.4% on Smoky.

4.4.2 S3D Performance

S3D is a state-of-the-art flow solver for performing direct numerical simulation (DNS) of turbulent combustion [44]. We use a modified version of S3D code called S3D_Box created by the S3D team for our test. S3D_Box performs a portion of the full S3D simulation. During its execution, S3D_Box periodically outputs species data which are 22 3-dimensional double-typed arrays. The species data is fed into a parallel volume

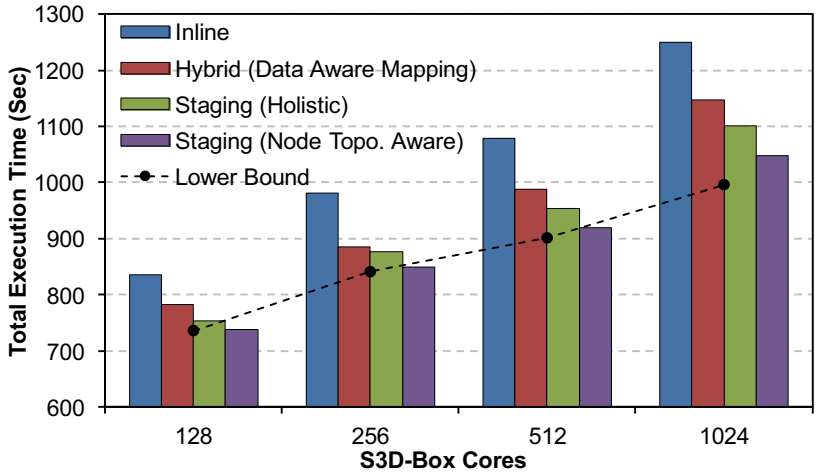
rendering code [152] to visualize images for each every species. The visualization code reads data by specifying global array index ranges and FlexIO handles MxN data re-distribution underneath. We set the input parameters so that during each I/O action, the total size of 22 arrays generated by each simulation process is 1.7MB, which is the same as typical production S3D runs. The simulation writes species data out every ten simulation cycles.

1) Tuning Data Movement.

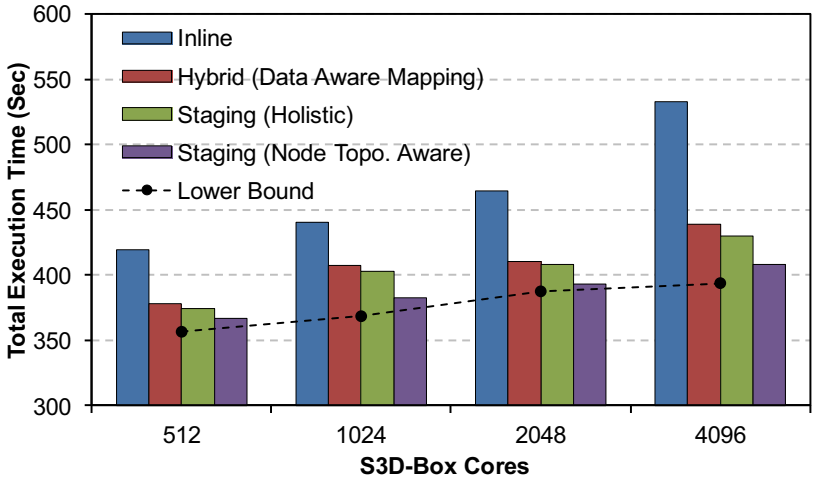
Data movement between simulation and visualization exercises FlexIO's Global-Array-oriented I/O pattern. Since arrays' distribution and memory addresses do not change over time at neither simulation nor visualization side, we set the caching option to CACHE_ALL to avoid several gather/scatter and handshaking messages during data movement (as described in Section 4.2.2). We also enable batching so that all 22 arrays are packed and sent together in a batch. Besides, simulation's write calls are set to be asynchronous. Those tuning efforts significantly reduce the simulation-visible data movement time on both Titan (from 1.2 to 0.053seconds when S3D_Box runs on 1K cores with RDMA transport) and Smoky (from 4.0 to 0.077seconds when S3D_Box runs on 1K cores with RDMA transport). And due to the small output data size, asynchronous data movement does not cause visible impact on simulation's internal communication. The tuning is enforced through setting hints in external XML configuration file and requires no changes to simulation or visualization source code.

2) Tuning Placement of Analytics.

We apply the three heuristic algorithms to decide placement of the visualization for S3D_Box. The resource allocation step determines a 128:1 ratio between simulation and analytics processes. For S3D case, the intra-program MPI communication volume is dominant over inter-program data movement due to relative small output data size and low I/O frequency. Under this situation, both Holistic Placement and



(a) Total Execution Time on Smoky



(b) Total Execution Time on Titan

Figure 24: S3D_Box performance tuning.

Node Topology Aware Placement place visualization processes onto separate nodes (i.e., Staging Nodes) and use RDMA transport to move data between simulation and visualization processes. They also place S3D_Box in a 3D block decomposed fashion to respect S3D_Box's logical 3D process layout. Node Topology Aware Placement achieves slightly better performance than Holistic Placement by further aligning processes' communication with compute node's NUMA structure (as shown in Figure 24).

The Data Aware Mapping algorithm places each analytics process close to those simulation processes which intensively communicate with it. This ends up placing visualization processes in a hybrid manner: a visualization process receives data from

simulation processes both on local node and on remote nodes. Since the inter-program data movement volume is much less than internal MPI communication, putting simulation and visualization close to each other does not pay off sufficiently, but meanwhile such hybrid placement increases the amounts of S3D_Box's MPI communication that goes across interconnect and increase Total Execution Time compared to the staging placement.

We measure the performance with inline placements. Staging placement is better than inline because asynchronous data movement and running simulation and visualization computation (and writing rendered image to files in PPM format) as a two-stage pipeline can effectively hide the cost of I/O and analytics computation. Due to insufficient scalability of file I/O, the advantage of staging placement over inline increase at larger scales. Staging placement also consumes less CPU hours than Inline, since it use 0.78% additional resources but improves Total Execution Time by up to 19% and 30% on Smoky and Titan, respectively.

3) How Close is Our Solution to the Optimal?

The runtime of S3D_Box when it runs solo and does not perform I/O or analysis gives the lower bound of the optimal Total Execution Time (dashed lines in Figure 24). With less than 1% extra resources, the staging placement is at most 3.6% larger than the lower bound on Titan, and 5.1% on Smoky.

In summary, for S3D, placing visualization on a set of staging nodes and aligning both inter- and intra-program data movement with underlying architecture gives the best performance and cost, and the savings of tuning placement is more evident at larger scales.

4.4.3 Utility of Data Conditioning Plug-ins

FlexIO's Data Conditioning Plug-in enables dynamic and flexible computation placement along I/O path. Applications can leverage this feature to implement effective

runtime policies to customize simulation output data on-the-fly, or adapt to dynamic variations in workloads or environment. We demonstrate the utility of DC Plug-ins with examples.

1) Dynamic Data Selection

A common practice for scientists to monitor simulation status is to let simulation periodically output a set of variables and run validation codes on those data. We implement an instance of DC Plug-in for GTS simulation with which validation code can specify the particle attribute(s) it wants to check, and this DC Plug-in can be dynamically deployed and run at either simulation or local analytics side. We use three data selection instances which select 1, 3, and 7 attributes from all 7 attributes of particles, respectively. On Smoky, we run GTS on 256 cores and the validation analytics on separate 32 cores. Figure 25 shows the measured simulation runtime and data movement volume between simulation and analytics when data selection plug-in is deployed at simulation side, and compares the results when all the original particle data are moved to validation analytics ("No Plug-in"). Deploying data selection plug-in with large data reduction ratio onto data source (simulation) can effectively reduce data movement, and cause negligible overhead to simulation blocking I/O time. In fact, reducing data movement volume also improves simulation runtime due to reduced contention on interconnect.

2) Load Shedding

If the analytics consumes data slower than simulation generates data, it will blocks simulation and may cause huge waste of CPU cycles at large scale. Under this situation, DC Plug-ins can be used to either shift workload from analytics to simulation or reduce data being moved downstream so that load on analytics side is alleviated. To demonstrate this, we implement a data staging service for GTS which asynchronously moves output data from simulation and dumps data to files. We emulate a situation where the file system is experiencing severe congestion so writing to file is very slow

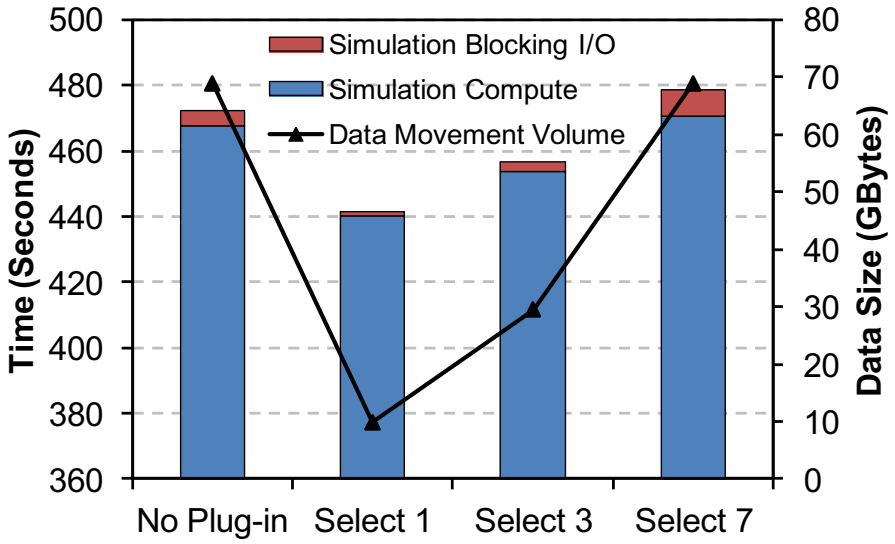


Figure 25: Performance impact of data selection plug-in to simulation.

(which does happen in practice) and causes back-pressure to simulation. To cope with this situation, the staging server instantiates a sampling DC Plug-in at simulation side which samples one out of every 100 particles of the original simulation output data. A simple policy is used to trigger load shedding: sampling plug-in is installed to simulation side if monitored simulation's running-average blocking I/O time exceeds a pre-defined threshold value. The dynamic code generation requires only 0.5msecs, so code deployment has an insignificant impact on the running system. The resulting sampling code requires only 220 x86 instructions. Figure 26 compares the steady state time before vs. after DC Plug-in is deployed. The sampling Plug-in helps reduce data fetch time and staging server's file writing time and releases GTS simulation from blocking.

To summarize, experiments show that FlexIO can support a variety of simulation and analytics workloads at large scales through flexible placement options, efficient data movement, and dynamic deployment of useful data manipulation functionalities.

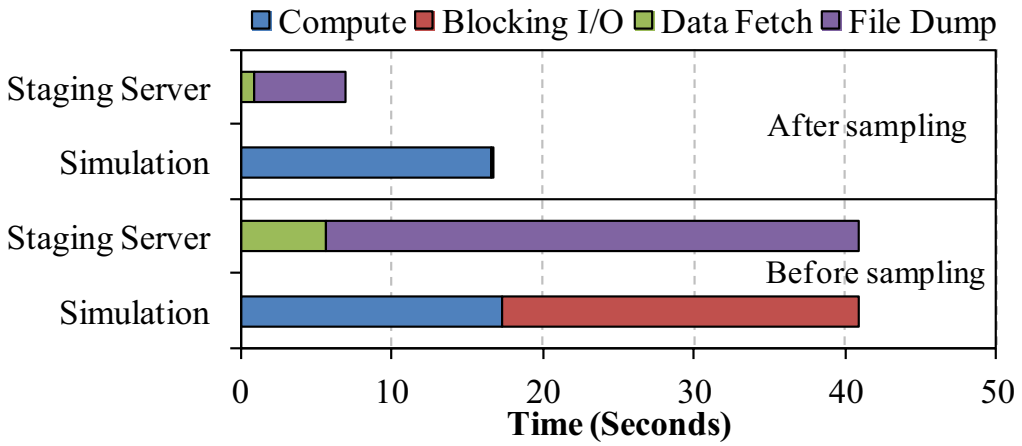


Figure 26: Load shedding to adapt to slow staging server. GTS runs on 128 cores and Staging server runs on 16 cores on Smoky.

4.5 Related Work

Online data analytics and visualization has gained much recent attention from the HPC community. Current work falls into two categories: (1) new data analytics and visualization algorithms, including in situ indexing [35, 69], compression [73], feature extraction [4], feature extraction [3], and various visualization techniques [151, 102], and (2) supporting tools and infrastructures like those mentioned in Introduction Chapter.

Computation placement is an extensively studied topic in distributed systems due to its significant impact on application performance and cost. Particularly relevant is previous work on computation placement within the Active Storage context. Abacus [18] uses an online performance model to guide the dynamic placement of application and file system functions among clients and servers to adapt to a variety of application and system runtime characteristics, but it assumes a progressive, per-record computation model. [147] studies load distribution of a class of streaming computation in an active storage system. Diamond [60] aggressively places filters to data sources to reduce search operation costs.

The importance of placement has also been exploited in other distributed computing models. Streaming operator placement on wide-area overlay network has been studied in [113]. COLA [68] applies graph partitioning to place a streaming processing dataflow onto a cluster of nodes with load balance and throughput as the major optimization objectives. Armada [105] uses similar graph partitioning techniques to distribute in-network operations within a Data Grid to improve I/O performance. Although the environments targeted by those work are different from the HEC platforms targeted by FlexIO, most placement algorithms can be supported by FlexIO thanks to its diverse placement options and performance monitoring information.

There have emerged many data intensive computing platforms such as IBM's System S streaming system [19], SciDB [37], and Hadoop/MapReduce-related systems (e.g., SciHadoop [26], Himach [136], and SciMATE [145]). Those are self-contained frameworks with specific programming models and built-in runtime to manage computation distribution. FlexIO as an I/O middleware is beneficial to those frameworks in that they may leverage FlexIO to couple with simulation for online data processing and enjoy the location-flexibility brought by FlexIO.

Scientific workflow systems like Pegasus [40] and Kepler [93] are often used to orchestrate the execution of analysis tasks. They mainly use files as the data exchange mechanism. The explosive growth of scientific data, however, can easily stress the I/O system and overwhelm overall workflow performance. Therefore, it is expected that more and more analysis will be deployed online and run in situ with simulation, especially those which can achieve early data reduction or prepare data for better use by downstream analyses. FlexIO can be readily integrated with scientific workflow systems to enable such online usage.

At the implementation level, our shared memory transport borrows cache optimizations from FastForward's lock-free queue [52]. There is also similar work on high performance MPI intra-node communication [27]. Besides, although MPI may be

used to achieve similar flexibility as FlexIO, MPI does not support seamless switch to file I/O, and the MxN code coupling tools built on top of it are shown to have efficiency issues for large data exchange [155].

4.6 Conclusions

The FlexIO middleware is designed to flexibly couple online data analytics with simulation on high end machines. Evaluation results obtained with two large scale scientific applications GTS and S3D verify the argument for flexible placement and demonstrates FlexIO's ability to support common I/O patterns and diverse placement options. In addition, various placement policies can be implemented with FlexIO to effectively tune application performance and cost. Finally, Data Conditioning Plugins enable dynamic deployment of computation along I/O path based on which useful runtime functionalities can be implemented.

CHAPTER V

EFFICIENT ANALYTICS USING NON-DEDICATED RESOURCES

5.1 Introduction

The research presented in this chapter has two goals: (1) to improve the resource efficiency of running online data analytics, and (2) to do so without perturbing the simulations running on the same nodes. In particular, we seek to over-subscribe compute nodes by co-locating simulation and analytics computations, without affecting the simulation execution, while at the same time, efficiently using compute node resources to run online analytics.

Measurements of six representative scientific simulations motivate the argument that node over-subscription can be cost neutral to the core simulation. Specifically, we demonstrate that the well-tuned MPI/OpenMP implementations of these codes written for high end machines leave substantial unused resources (CPU and memory) on compute nodes, which can then be used to run online analytics. One cause is sequential periods in these codes (i.e., when the execution flow is outside their OpenMP parallel regions) in which worker threads wait on the MPI process' main thread. Although most such sequential periods are short, their aggregate duration can be up to 65% of total execution time in these real-world codes.

Previous work has sought to reduce sequential periods and utilize spare node resources by overlapping the main thread's sequential work with OpenMP regions, but such application-specific tuning efforts are limited by data and control dependencies, and they can also impede code clarity and portability. In fact, none of the six codes in our study uses such overlapping in their production versions. The novel

"GoldRush" method presented in this chapter uses a different approach to exploiting idle node resources: it uses them to run the online data analytics needed to cope with I/O bottlenecks. Benefits include the efficient use of compute node resources and reductions in data movement overheads, as will be demonstrated with detailed performance measurements. The GoldRush method is made possible by the FlexIO transport (presented in Chapter IV) in the ADIOS I/O system [88] widely used on high end machines. Specifically, with FlexIO and ADIOS, analytics pipelines can be configured to map to compute nodes only those portions of their computations that "fit into" available idle resources, with additional analytics mapped to dedicated resources and/or run as post-processing tasks after data has been moved to the machine's attached parallel file system. Appropriate end-to-end mappings of analytics pipelines can reduce I/O data volumes and data movement overheads [162, 1], to provide science end users with rapid insights into the data produced by their simulations.

Leveraging such flexibility in constructing data analytics pipelines, this chapter addresses the key compute-node-level challenges for efficiently running online data analytics. The first challenge is that for well-tuned scientific simulations, idle compute-node CPU cycles exist in the form of a large number of short idle periods. This makes it difficult to schedule and allocate cores to analytics without causing undue runtime overheads for the simulation. Second, because co-located simulation and analytics codes share certain node resources (e.g., last level caches, memory busses and controllers), the execution of analytics must be managed to minimize the degrees to which simulations are perturbed. Measurements presented in this chapter demonstrate that carefully managing how analytics are run is critical to achieving overall high performance for co-located simulation and analytics. Third, current operating systems on HEC platforms are not well equipped to deal with multi-programmed simulation and analytics workloads, as they schedule processes based on core idleness,

essentially allocating idle resources to analytics in a greedy manner, and they are also largely ignorant of potential interference effects. Therefore, even with carefully configured process priorities, such policies can lead to severe performance loss. As shown later, priority-based OS level scheduling of analytics processes can result in an up to 57% performance degradation of the simulations.

To address those challenges, we have created a lightweight runtime system, named "GoldRush", which supports resource-efficient and non-intrusive online data analytics. GoldRush (i) uses low-overhead online monitoring to identify opportunity windows during which (ii) it can schedule analytics to run on cores not currently used by the simulation. It also (iii) continuously assesses interference between concurrently-running simulation and analytics, and (iv) controls the execution rate of analytics processes to mitigate harmful impacts on the simulation due to contention on shared node resources.

GoldRush makes the following contributions:

1) Fine-Granularity Operation: during simulation execution, it identifies idle periods, predicts the duration of each period, selects those periods with sufficient durations to run analytics, but skips those that are too small to dwarf context switching overheads. It completely suspends analytics when cores are in use by the simulation, to avoid perturbing the parallel simulation.

2) Interference Awareness: it can detect interference between concurrently running simulation and analytics arising from contention on shared memory resources, and it dynamically mitigates such interference by throttling the execution rate of analytics.

3) Low Overhead: runtime overheads (including monitoring and scheduling) are negligible, measured as never exceeding 0.3% of total runtime with representative HEC applications.

4) Transparency: its methods are easily integrated into existing HEC runtimes, demonstrated by their use with OpenMP/MPI hybrid codes, thus imposing minimal

restrictions on current simulation and analytics codes. By effectively managing co-located simulation and analytics workloads, GoldRush complements existing online data analytics techniques [2][6][7][42][46], opening up new opportunities to efficiently run such analytics without the need to dedicate compute node resources, leading to substantial performance improvements and cost savings at large scales.

GoldRush is evaluated with real-world scientific applications on NERSC's Hopper Cray XE6 and Oak Ridge National Laboratory' InfiniBand cluster. In particular, measurements with co-located simulation and synthetic analytics show that GoldRush's synergistic scheduling improves simulation performance by 9.9% on average (and up to 42%) over the OS scheduling. For a fusion application GTS, there is a clear trend that GoldRush's advantage over the OS baseline native scheduling methods increases at larger scales (up to 7.5% at 12288 cores); and that the GoldRush-managed analytics outperforms alternative analytics setups: for GTS at 12K cores, it achieves 30% performance improvement over "Inline" analytics and a 1.8x reduction in data movement volumes over "In-Transit" analytics. Additional evaluations on a 32-core, multi-socket Intel Westmere machine demonstrate GoldRush's node-level scalability and applicability across different architectures.

The remainder of this chapter is organized as follows. Section 5.2 motivates GoldRush with experimental measurements that show the benefits and challenges of leveraging idle compute node resources for online data analytics. Section 5.3 describes the system design and implementation of GoldRush and the techniques used to gain high levels of performance and resource efficiency. Section 5.4 evaluates GoldRush with both synthetic benchmarks and real-world applications on different HEC platforms. Section 5.5 reviews related work and Section 5.6 draws conclusions.

5.2 Motivation

This section presents a detailed characterization of the idle resources on compute nodes, to quantify the potential benefits and challenges of using them.

5.2.1 2.1 Characterizing Idle Resources

Figure 27 illustrates the execution of a MPI process with multiple OpenMP threads. When only the main thread in the MPI process is actively executing some sequential code outside OpenMP regions (i.e., in sequential periods), the OpenMP worker threads are waiting and the cores on which they run become idle ("P1" to "P6" in Figure 27). Typical sequential periods involve MPI communications, file I/O, and/or non-parallelized computations. Analytics can be run asynchronously, in response to a simulation's data output action and using available idle cores, as long as there is sufficient free memory for buffering output data between successive simulation output actions.

We are interested in how many idle resources (CPU and memory) exist when running real-world HEC simulation codes and whether those idle resources are amenable for use by online analytics. Toward that end, we profile four widely-used and well-tuned MPI/OpenMP hybrid simulation codes: GTC (fusion) [70], GTS (fusion) [139], GROMACS (molecular dynamics) [2], LAMMPS (molecular dynamics) [114], plus two well-known MPI/OpenMP hybrid benchmark codes: BT-MZ and SP-MZ from the NPB benchmark suite [8].

The six codes are profiled on NERSC's Hopper Cray XE6 [7] and on ORNL's Smoky InifiniBand cluster [49]. Hopper has 6,384 compute nodes and uses Cray's Gemini interconnect. Each Hopper compute node has two 12-core MagnyCours AMD processors. There are 4 NUMA domains, each with 6 cores and 8GB DRAM. Smoky is an 80 node cluster, where each compute node has four quad-core AMD Opteron processors. There are 4 NUMA domains, and each domain has 4 cores and 8GB

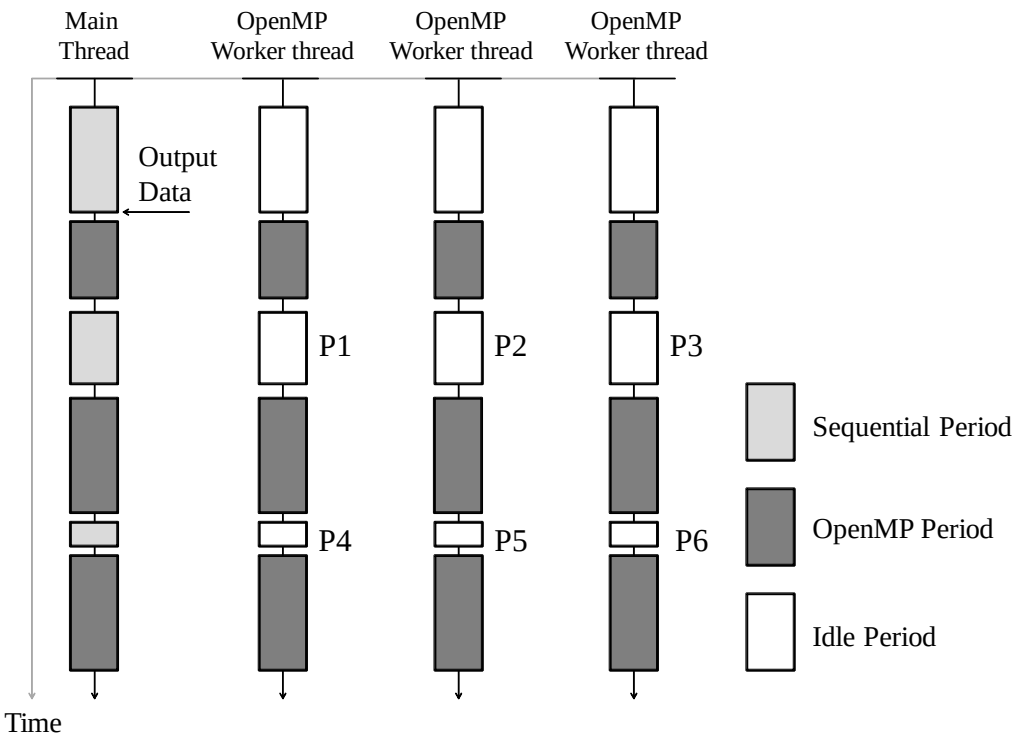


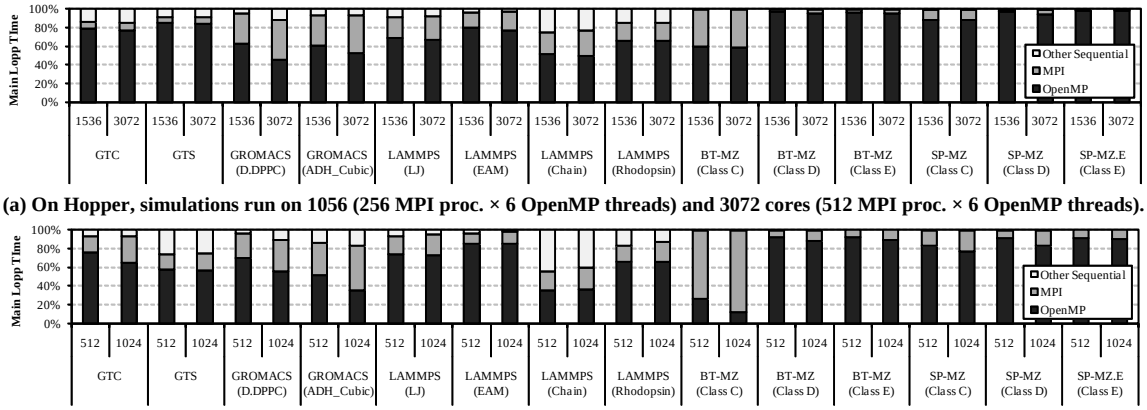
Figure 27: Illustration of idle resources during execution of a MPI processes with 4 OpenMP threads. The 3 OpenMP worker threads are idle when the main thread is in sequential periods.

DRAM. To accommodate the NUMA architecture, we run each MPI process in one NUMA domain and run as many OpenMP threads as the number of cores in each NUMA domain (which leads to peak performance for all simulation codes). Threads are pinned on cores, and memory affinity is enforced within each NUMA domain with the aprun and mpirun launch facility.

GTC, GTS, BT-MZ and SP-MZ are built with the PGI compiler, and GROMACS and LAMMPS with the GCC compiler, respectively (as suggested by the developers). Codes are run with representative input configurations, and GROMACS, LAMMPS, BT-MZ, and SP-MZ are run with the multiple input decks distributed with these software packages. The CrayPAT [36] and Vampir [9] tools are used to collect profiling information.

Each simulation's main loop time is divided into three parts: (1) OpenMP periods (all threads are active), (2) MPI periods (only the main thread is active, performing MPI communications), and (3) "Other Sequential" periods (only the main thread is active, carrying out sequential activities like file I/O or others). In the latter two cases, the cores on which OpenMP worker threads run are idle. Figure 28 shows the percentages of execution time spent in those three parts.

Interesting observations from these measurements include the following. First, jointly, all idle periods (MPI and Other Sequential periods) comprise up to 65% of the total main loop time for four of these applications (i.e., LAMMPS with the "Chain" input deck), and even 89% for the NPB BT-MZ benchmark with the class C input. Note that on Hopper's compute nodes, 20 out of 24 cores are idle during those periods, leading to substantial amounts of idle compute capacities. Second, the percentage of total idle periods generally increases when scaling the simulation to run on more cores. For example, GTC's idle period percentage increases from 21% to 23% when scaling from 1536 to 3072 cores on Hopper. This holds for weak scaling codes like GTC, GTS, and LAMMPS in which MPI communication times increase at larger scale,



(a) On Hopper, simulations run on 1056 (256 MPI proc. × 6 OpenMP threads) and 3072 cores (512 MPI proc. × 6 OpenMP threads).

(b) On Smoky, simulations run on 512 (128 MPI proc. × 4 OpenMP threads) and 1024 cores (256 MPI proc. × 4 OpenMP threads).

Figure 28: Breakdown of simulation main loop time. The input decks are specified in parentheses following the simulation names. When the simulation is in non-threaded sequential periods, only its main thread is active and OpenMP worker threads are idle.

and also for strong scaling codes like GROMACS and the NPB benchmarks, where in OpenMP times decrease with increased core counts. Third, although simulation performance varies across inputs (like LAMMPS and GROMACS), it is common that idle periods comprise a substantial portion of total simulation runtime.

We also measure peak memory usage among all MPI processes. None of the simulation codes consume more than 55% on either Hopper or Smoky. The resulting available free memory makes it feasible to buffer simulation output data, thereby enabling the asynchronous execution of analytics and simulation codes.

5.2.2 Challenges of Using Idle Resources

Although the measurements shown so far demonstrate sufficient availability of idle resources, there are several challenges for effectively harvesting these idle resources for online data analytics, discussed next.

5.2.2.1 Magnitude of Idle Resources

Despite the substantial amounts of total idle CPU cycles, most individual idle periods are short in duration. Figure 29 shows the distribution of durations of idle periods

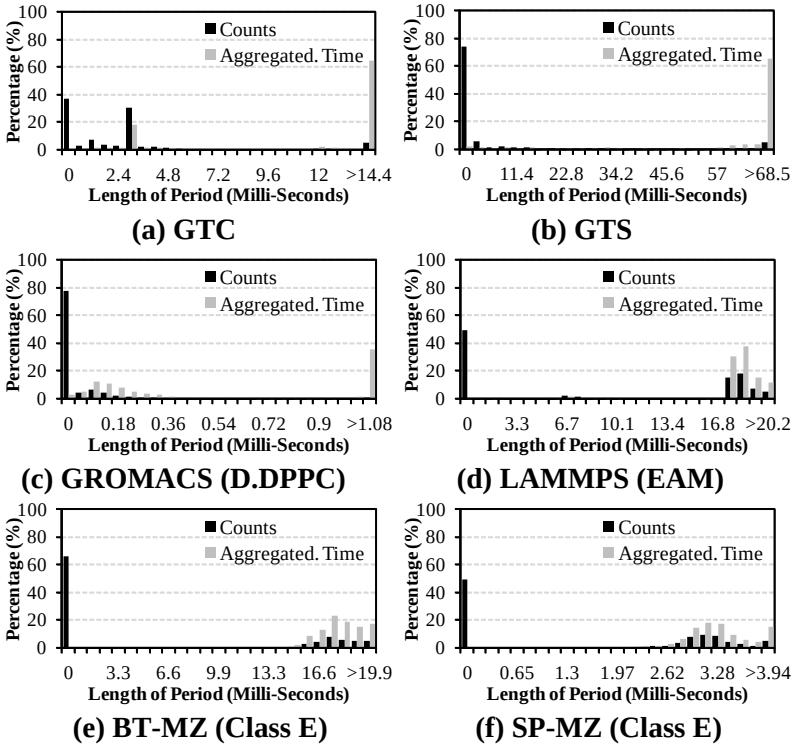


Figure 29: Distribution of idle period duration. All simulations run with 1536 cores on Hopper.

in our six codes. The "Count" histograms show that for all simulation codes, the majority of idle periods are quite short (less than 1ms), while the "Aggregated Time" histograms show that the total amount of idle time is dominated by a modest number of large idle periods.

This distribution pattern has important implications. First, it is not likely useful, in terms of cost vs. benefit, to harvest small idle periods. As a result, one must determine, at runtime, which idle periods will be sufficiently large to warrant their use for running desired data analytics. Second, inaccurate methods for identifying appropriately long idle periods will lead to inefficiencies for two reasons: (1) insufficient benefits or worse, undue overheads when using periods that are too small, and (2) missed larger periods leading to loss of major portions of total available idle time.

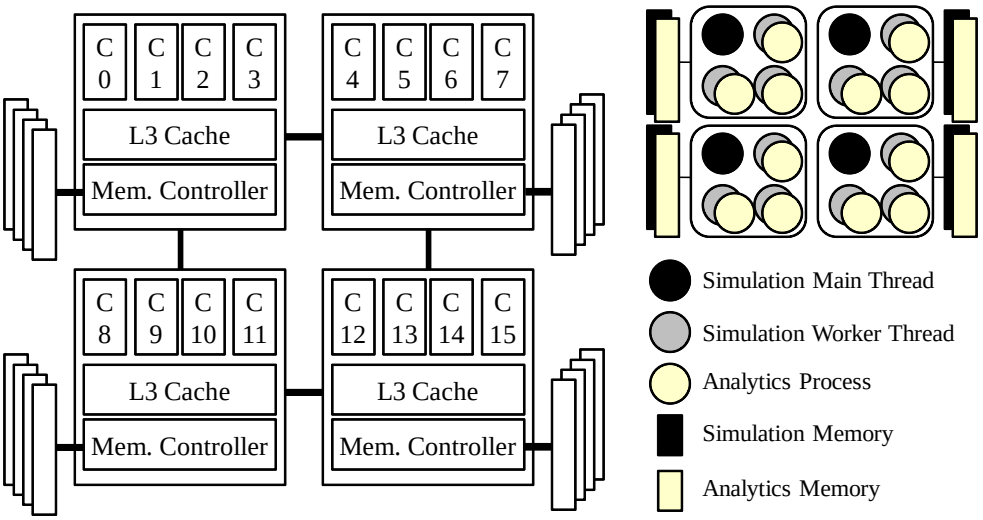


Figure 30: Placement of simulation and in situ data analytics on Smoky's 16-core compute node.

5.2.2.2 Contention on Shared Resources

Beyond finding idle periods suitable for running analytics, another issue is the potential interference of analytics imposed on the simulation's main thread running in its sequential phase (during which analytics processes concurrently run on idle cores not used by the simulation's OpenMP worker threads). Interference is due to contention on resources shared between both sets of threads, such as the last level cache, the memory bus, and the memory controller (as shown in Figure 30); it is particularly harmful for tightly synchronized parallel simulations, as the slowdown of each individual MPI process may cascade and be amplified when running at larger scales [59].

5.2.2.3 Limitations of Operating System Scheduling

A baseline solution for co-running analytics with simulation threads is to leave it to the Linux OS scheduler and the OpenMP runtime to manage both workloads. We realize this approach as follows.

1) On each compute node, fork some number of analytics processes. Set their CPU affinities so that they can run on the cores where the simulation's OpenMP

Table 2: Analytics benchmarks.

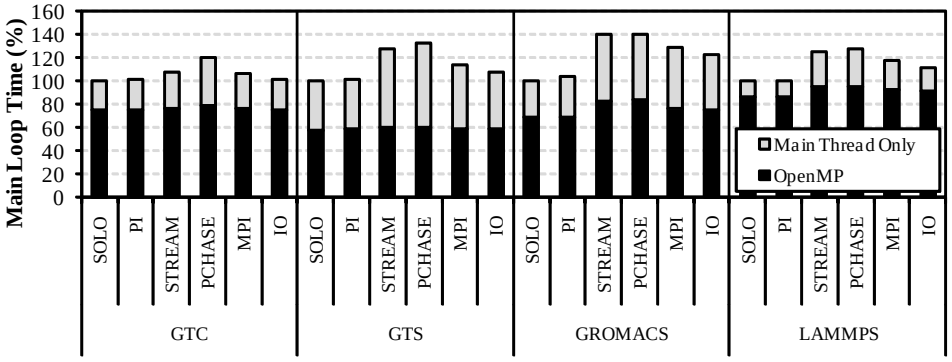
Benchmark	Tasks for Each Program
PI	Iteratively calculate Pi.
STREAM	Traverse randomly linked lists (200MB in total).
PCHASE	Sequentially scan large arrays (200MB in total).
MPI	Collectively call MPI_Allreduce() on 10MB data.
IO	Write 100MB data to parallel file system.

worker threads are run, but not on the cores hosting the simulation's main threads. The analytics processes are given the lowest priority (with "nice" values set to 19).

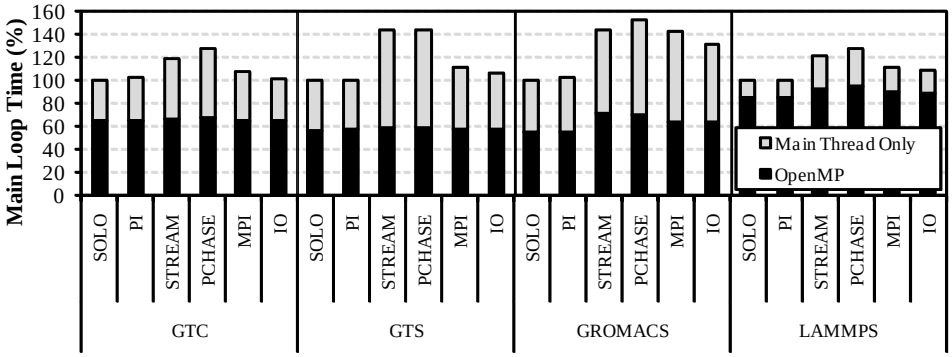
2) Configure the simulation's OpenMP runtime so that worker threads yield CPUs when they are outside OpenMP regions. For the Intel OpenMP runtime, this can be achieved by setting the KMP_BLOCKTIME environment variable to 0. The PGI and GNU OpenMP runtimes can be similarly configured, by setting the OMP_WAIT_POLICY environment variable to "PASSIVE". The priorities of the simulation's OpenMP worker threads are set to default (their "nice" values are equal to 0).

This baseline solution is evaluated by co-running the six simulations with the five analytics benchmarks listed in Table 2. These benchmarks each stress a certain subsystem in the machine. On Smoky, we run each simulation with 512 cores (128 MPI processes and 4 OpenMP threads per process) and with 1024 cores (256 MPI processes, each with 4 OpenMP threads). In both cases, there are 16 simulation threads and 12 analytics processes on each compute node, as shown in Figure 30.

Figure 31 shows the performance of four simulations with co-running analytics. Each simulation's main loop time is divided into two parts: parallel OpenMP periods and Main-Thread-Only periods (the latter correspond to MPI and Other Sequential periods in Figure 28). With the pure OS-based management solution, co-located analytics slow down simulations by up to 57% compared to simulations' solo runs, and performance degradation generally becomes worse at larger scales.



(a) Simulation main loop time with 512 cores on Smoky.



(b) Simulation main loop time with 1024 cores on Smoky.

Figure 31: Simulation performance with co-located analytics.

The ineffectiveness of pure OS-based management is caused by several factors. First, the significant slowdown of the Main-Thread-Only periods shown in Figure 31 indicates that the simulation's main threads experience severe interference from concurrently running analytics. This is particularly true for cases in which the simulation's main threads co-run with memory intensive codes like PCHASE and STREAM, because those benchmarks cause severe contention on the last level cache, memory controller, and other shared resources in the memory hierarchy. Linux' default OS scheduler does not recognize those facts, as its main focus is on core idleness.

Second, there are increases in some simulations' OpenMP times with the presence of co-located analytics. One reason is the OS scheduler's greedy nature, which always schedules analytics threads as soon as the OpenMP worker threads yield the CPU. For short idle periods, analytics threads will be forced to suspend soon after they begin to run, to return cores back to higher priority simulation threads. Another reason is the Linux scheduler's imposition of fairness on analytics vs. simulation threads, causing it to allocate time slots for, rather than completely suspend, low-priority analytics processes while the simulation's worker threads are active (i.e., in a parallel OpenMP period). This causes jitter to the simulation and negatively impacts its performance.

The GoldRush runtime methods described next remedy these shortcomings of the OS baseline solution.

5.3 GoldRush Runtime System

5.3.1 Overview

GoldRush manages the execution of data analytics co-located with simulation processes, in ways that (i) leverage unused idle resources on compute nodes, and (ii) mitigate potential interference between simulation and analytics. GoldRush is implemented as a runtime library and residing at both the simulation and analytics sides of these compute node-based computations (highlighted in yellow in Figure 32).

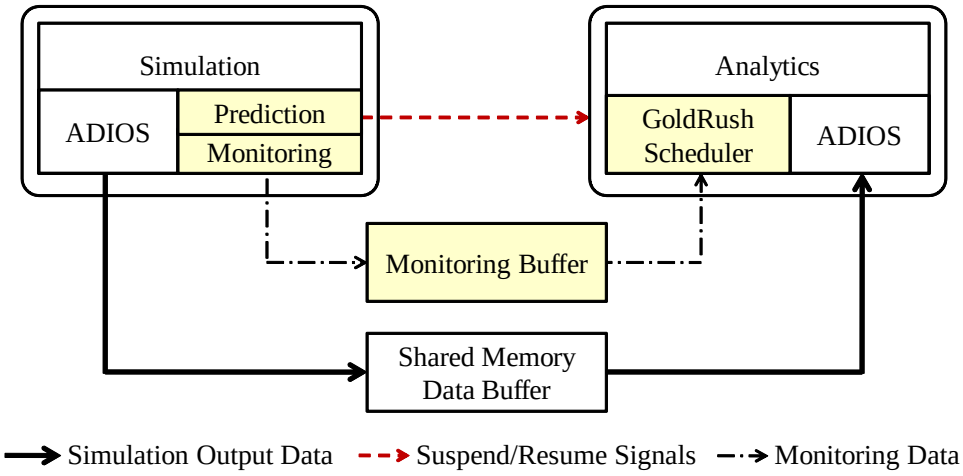


Figure 32: Architecture of GoldRush runtime.

For simulation processes, GoldRush generates performance monitoring metrics used by a prediction module to estimate the lengths of upcoming idle periods at the exit of each OpenMP parallel region. If the next idle period is predicted to be "usable", GoldRush sends signals to analytics processes to resume their execution; if no signal is produced, analytics processes remain suspended throughout the next idle period. Once resumed, analytics processes run on the cores yielded by the simulation's OpenMP worker threads, while the simulation's main threads continue to run on their own, dedicated cores. When the simulation's main threads reach the end of their idle periods (i.e., the start of next parallel OpenMP region), signals are sent to suspend analytics processes, thereby permitting the simulation's OpenMP worker threads to re-gain exclusive use of their cores for executing the subsequent parallel OpenMP period.

To assess potential interference between simulation and analytics, the GoldRush runtime also periodically updates a shared memory monitoring buffer with performance data about the simulation's main threads. The analytics-side GoldRush scheduler periodically reads this information, assesses interference severity and if significant interference is detected, the scheduler throttles, i.e., slows down, the execution rate

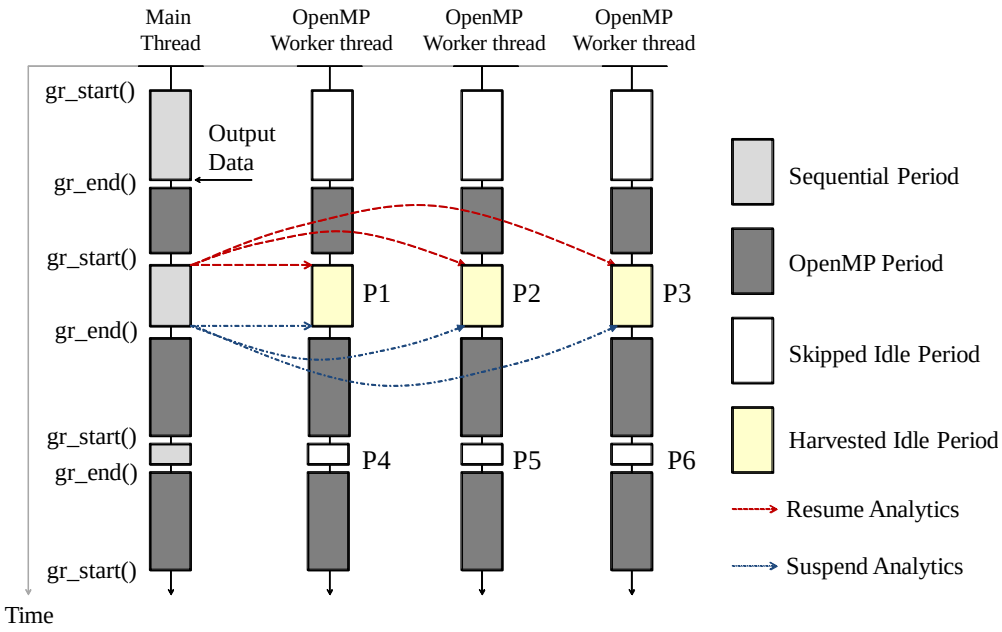


Figure 33: Simulation and analytics execution timeline.

of analytics processes. This serves to reduce contention on shared resources, at the cost of reduced progress with analytics processing. A limit on possible slowdown is imposed by the fact that analytics processing must be completed before the simulation's next output steps are taken. On-compute-node analytics, therefore, have to be "sized" appropriately, and we do so by leveraging the placement flexibility offered by the ADIOS IO library and its FlexIO IO methods, described in Chapter IV. With ADIOS and FlexIO, analytics pipelines can be defined and (re-)structured to match available compute node resources, with "overflow" analytics actions performed in separate "staging nodes" reserved for online analytics and/or postmortem, after data has been moved to disk. Another attribute of the FlexIO transport used by GoldRush is its efficient intra-node data movement from simulation to analytics via a shared memory transport.

Compared to the baseline solution described in Section 5.2, GoldRush adds potential overheads to the simulation side for performance monitoring and idle period prediction, and for suspending and resuming analytics. There are also additional

Table 3: GoldRush public API.

Function	Description
int gr_init (MPI_Comm comm);	Initialize the GoldRush runtime
int gr_start (char *file, int line);	Mark the start of an idle period
int gr_end (char *file, int line);	Mark the end of an idle period
int gr_finalize ();	Finalize the GoldRush runtime

costs at the analytics side for online monitoring and execution control. As shown in Section 5.4, these overheads are negligible, permitting GoldRush to significantly improve application performance and resource efficiency over the baseline solution.

5.3.2 Inter-Posing GoldRush

GoldRush is implemented as a C library, for which we offer two approaches to integrating it with simulation codes. The first approach directly inserts the GoldRush API (listed in Table 3) into the simulation's source code. In particular, a gr_start() call is placed at the end of an OpenMP code region (e.g., after a "!$omp end parallel" statement) to mark the start of an idle period; and a gr_end() call is put before the beginning of an OpenMP parallel region (e.g., before a "!$omp parallel" statement) to mark the end of an idle period. At runtime, those markers are executed by the main thread of each simulation process to identify the beginning and end of idle periods, and to perform operations that monitor performance and resume/suspend analytics processes.

The second approach integrates the library with the simulation in a more transparent fashion, avoiding changes to simulation codes, by adding its functions into appropriate routines within the OpenMP runtime library. As a proof of concept, we modify GCC's libgomp runtime library by instrumenting the runtime routines associated with PARALLEL and FOR directives. Those are sufficient to cover all of the top-level OpenMP regions in the GTC, GTS, LAMMPS, GROMACS, and NPB codes. Other directives can be supported similarly, left for future work.

In comparison, the source code instrumentation approach is more general and

flexible at the cost of manual code modification. The instrumented OpenMP runtime library approach is transparent to simulation codes, but requires modifying internals of the OpenMP library. In practice, we have instrumented the sources of simulation codes requiring Intel or PGI compilers, as those compilers' OpenMP runtime libraries are not available to us for modification. Besides, source instrumentation may be automated with source transformation [98] or binary re-writing tools [21].

Analytics codes only need to add gr_init() and gr_finalize() functions, permitting an instance of the GoldRush scheduler to be activated in each analytics process at runtime.

5.3.3 Online Monitoring and Prediction

5.3.3.1 Predicting Idle Period Durations

At the beginning of an idle period (i.e., in a gr_start() call), the simulation's OpenMP worker threads have yielded their cores, and the main thread is about to enter a sequential code region. An important decision to make at this point is: should the analytics processes be allowed to run on idle cores during this upcoming idle period? As discussed in Section 5.2.2.1, idle periods are appropriate only if they are sufficiently long. To predict their expected durations, the GoldRush runtime records the timings and number of occurrence of each executed idle period. Each idle period is uniquely identified by its start and end locations (the file name and line number arguments passed to marker API calls). When a gr_end() marker is executed, the idle period that just completed is identified. The duration of that idle period is measured as the elapsed time between the two successive gr_start() and gr_end() calls made by the main thread. The online history maintains a running average duration and occurrence counts for each unique idle period seen so far.

We currently use a simple heuristic to predict idle period duration, using the above online history information. The method has high accuracy and low overheads for simulations with strong locality and regularity in their execution flows (a typical

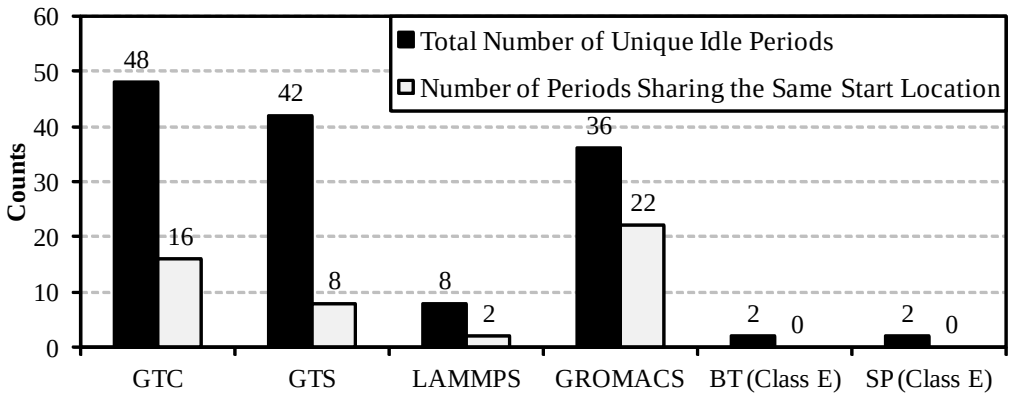


Figure 34: Number of unique idle periods and idle periods with the same start location (due to branching in execution flow).

behavior for many scientific codes), as those codes usually have a small number of unique idle periods with small variations in idle period duration. The heuristic works as follows. During the execution of gr_start(), a prediction function is called. It first finds all idle periods in the history that match the start location (file name and line number) of the upcoming idle period, selects the one with the highest occurrence count, and uses its running average duration as an estimate of the upcoming period's duration. If the estimated duration is greater than a pre-defined, tunable threshold value or no matching history record is found, the upcoming idle period is considered as "usable" for analytics.

Costs: The time and space costs of idle period prediction are proportional to the number of unique idle periods in a simulation's execution flow. As shown in Figure 34, the numbers of unique idle periods in the six simulation codes range from 2 to at most 48, resulting in low runtime overheads.

Prediction Accuracy: The purpose of prediction is to decide whether an idle period is usable (long) or not (short) with respect to a threshold value. Therefore, instead of using the absolute error in predicted duration values, we define a prediction of an idle period to be "accurate" if the predicted usability (short or long) of the idle period matches the indication of the actual duration. Specifically, we divide prediction

Table 4: Prediction accuracy with 1ms threshold (1536 cores on Hopper).

Simulation	Predict Short	Predict Long	Mispredict Short	Mispredict Long
GTC	31.6%	57.1%	6.4%	4.9%
GTS	58.5%	36.8%	3.6%	1.1%
LAMMPS	49.7%	49.7%	0.3%	0.3%
GROMACS	99.6%	0.1%	0.1%	0.2%
BT-MZ	66.6%	33.4%	0.0%	0.0%
GROMACS	50.1%	49.9%	0.0%	0.0%

results into four categories: (i) "Predict Short": correctly predict a short period to be short (not usable for analytics); (ii) "Predict Long": correctly indicate a long period to be long (usable); (iii) "Mispredict Short": wrongly predict a short period to be long; and (iv) "Mispredict Long": wrongly predict a long period to be short.

To quantify prediction accuracy, we record the predicted duration at the beginning of each idle period, and measure the actual duration at the end of the period, based on which we then count the number of predictions falling into each of the four categories described above. Table 4 presents the percentages of the four categories among all predicted periods, using a threshold value of 1ms. Accurate predictions range from 88.7% 100% of all predictions for the six simulations, showing that our prediction method is highly accurate for codes with regular execution flows.

Figure 35 shows how sensitive prediction accuracy is to the threshold value. When varying the threshold value from 0.1 to 2 milliseconds, prediction accuracy for all six simulations never falls below 84.5%, and remains 100% for BT-MZ and SP-MZ cases. Figure 35 also shows that 1ms is an appropriate threshold value since it leads to high accuracy and in addition, ensures that the selected usable periods are sufficiently large to amortize context switch overheads.

Despite good results with the six simulation codes used in our work, there remain substantial opportunities for future improvements and optimizations of methods for idle period prediction. For instance, for codes with dramatically varying idle periods

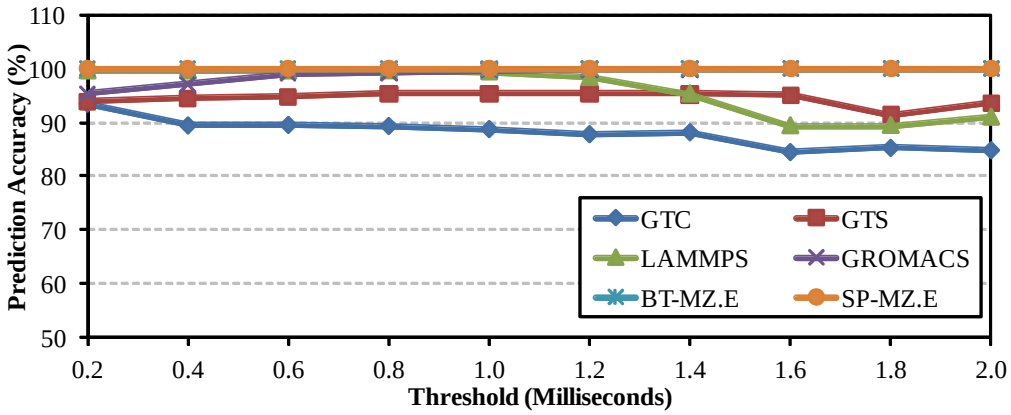


Figure 35: Sensitivity of prediction accuracy to the threshold value (Measured with 1536 cores on Hopper).

and runtimes (e.g., Adaptive Mesh Refinement codes), more sophisticated methods like dynamic call stack tracking plus statistical forecasting are likely preferable, which we will investigate as future work.

5.3.3.2 Monitoring Interference during Idle Periods

To manage potential interference between a simulation's main threads and concurrent analytics processes, GoldRush installs a timer and signal handler on each main thread to inspect relevant hardware performance counter values through the PAPI performance counter library [4], done every millisecond during idle periods. Measured are the number of CPU cycles and retired instructions, and IPC (Instructions per Cycle) is calculated to quantify the performance of the simulation's main thread. The IPC value is written to a per-simulation-process buffer in shared memory, and is periodically read by the analytics-side GoldRush schedulers. The timer is disabled at the end of each idle period.

5.3.4 Controlling Execution of Analytics

Analytics are run when an idle period is predicted as usable. This involves the simulation main thread sending a SIGCONT signal to resume the execution of analysis processes. Conversely, when the simulation main thread calls gr_end() at the end of

the idle period, it sends a SIGSTOP signal to suspend analytics. Analytics threads, therefore, are run only during selected idle periods; they are quiescent when the simulation is in its OpenMP regions. The signaling costs incurred are small (see Section 5.4).

An alternative to using signals to suspend and resume analytics processes is to set the simulation processes to use a real-time scheduling policy via the sched_setscheduler() system call. However, this privileged feature is not generally available in HPC environments (e.g., Hopper and ORNL's Titan Cray XK7).

5.3.5 Scheduling Analytics

At the analytics side, the GoldRush scheduler regulates the execution of analytics processes to mitigate potential interference effects experienced by the simulation's main threads. The scheduler is implemented as a signal handler in each analytics process and is periodically triggered by a timer. Two scheduling policies are presented below.

5.3.5.1 Interference-Aware Policy

The Interference-Aware scheduler works in three steps.

1) Assessing the Severity of Interference: once triggered, the scheduler reads the IPC value of the simulation's main thread from the shared memory monitoring buffer. Interference is determined as IPC being lower than some threshold value, whereupon the scheduler enters the next step; otherwise, the signal handler returns and analytics process runs at full speed until the next scheduling point.

2) Identifying Contentious Analytics Processes: each GoldRush scheduler instance determines whether the local analytics process to which it belongs is contributing to interference. Toward that end, it uses the L2 Cache Miss Rate (L2 Cache Misses per Thousand Cycles) as the indicator for the analytics process' contentiousness. If this miss rate is greater than some threshold value, then the analytics process

is subject to execution rate throttling. This is because an analytics process with high L2 Cache Miss Rate is likely to impose pressure on the shared L3 cache and on other shared resources, such as memory controllers and memory bus bandwidth.

3) Throttling the Execution Rate of Analytics: the scheduler throttles an offending analytics process' execution rate by putting it to sleep for some short period of time, by calling the usleep() function. When the sleep duration is exceeded, the scheduler's signal handler returns. The analytics then runs at full speed until the signal handler is triggered again, repeating the three scheduling steps. Since sleep duration controls the amount of idle cycles not used by analytics, the duration's value along with the scheduling interval used jointly provide useful knobs for controlling the percentages of idle cycles being harvested.

5.3.5.2 Greedy Policy

Under the Greedy policy, the analytics-side scheduler is disabled so that analytics processes run at full speed for all idle periods selected by the simulation-side prediction module. This policy differs from the OS baseline solution in that it relies on simulation-side prediction to filter out short idle periods. Comparing this Greedy policy with the Interference Aware policy and the baseline solution helps isolate the effects of simulation-side prediction and analytics-side scheduling.

5.3.6 Usage of GoldRush

GoldRush makes it feasible to deploy online analytics onto compute nodes so that useful analytics can run on otherwise-wasted idle resources, close to the data source (simulation), and in parallel with the simulation. It can improve the performance and/or resource usage of scientific applications' online analytics and I/O pipelines. A sample usage of GoldRush is to run as much analytics work on idle resources as the idle capacity permits, so that the amount of dedicated resources (e.g., dedicated cores [43] or staging nodes [160]) for online analytics can be reduced or even avoided.

Another usage is to perform data-reduction analytics operations with idle resources in compute nodes to reduce downstream data movements along the I/O pipeline.

5.4 Performance Evaluation

This section's experimental evaluations have three purposes: (i) analyze the cost and benefit of GoldRush runtime and its advantages over the OS baseline solution; (ii) measure the improvement of application performance and resource efficiency achieved by GoldRush for real-world applications; and (iii) assess the scalability of GoldRush with increasing machine size and node core count. The experiments are conducted on NERSC's Hopper Cray XE6 and ORNL's Smoky cluster.

5.4.1 Benefits of Synergistic Scheduling

Our first set of experiments co-runs simulation with "unrelated" analytics (the analytics does not operate on simulation output but on its private data set) under different scheduling policies. Here we evaluate scenarios where there is interference between the simulation and analytics. Note those are less likely to occur with "related" analytics in which there is cache-friendly, constructive data sharing between simulation and analytics – due to producer-consumer data reuse relationships. The purpose of these experiments is to assess GoldRush's ability to mitigate destructive interference between simulation and analytics. We co-run the four simulation codes (GTC, GTS, GROMACS and LAMMPS) with the five synthetic analytics benchmarks in Table 2. The simulation and analytics are set to run in four different configurations:

Case 1 (Simulation in Solo): Simulation is run without analytics; OpenMP worker threads do busy waiting in idle periods.

Case 2 (OS Baseline Solution): Simulation and analytics are co-located; OS schedules analytics processes to run whenever simulation's OpenMP worker threads yield CPUs.

Case 3 (Greedy Scheduling): Simulation-side GoldRush runtime selects idle

periods, resumes and suspends analytics with signals; Analytics-side GoldRush scheduler is disabled.

Case 4 (Interference Aware Scheduling): Simulation-side GoldRush selects idle periods to run analytics, resumes and suspends analytics, and also records simulation main threads' IPC values in shared memory buffer during idle periods. Analytics-side GoldRush scheduler does interference detection and control.

Simulations and analytics are run with 1024 cores on the Smoky cluster. They are placed in compute nodes as shown in Figure 30.

5.4.1.1 Benefits of GoldRush

Figure 36 shows the simulation's main loop time in the four cases. GoldRush with its greedy policy can improve the performance of the four simulations over the OS baseline solution. This demonstrates the importance of selecting proper idle periods at the simulation side. GoldRush with its interference aware policy can further improve simulation performance over the greedy policy, resulting in 9.9% on average and up to 42% performance improvement over the OS baseline solution. Figure 36 shows that such improvements are due to the reduction of the "Main-Thread-Only" portion of the main loop time. The difference of simulation run time in solo vs. under interference aware scheduling is at most 9.1% (GROMACS running with PCHASE) and 1.7% on average among all test cases, meaning that the simulations' performance is close to the optimal. These results demonstrate that GoldRush's interference aware scheduling can mitigate potential interference effects between the simulation's main threads and analytics processes during idle periods. Such advantage is the most evident for memory intensive benchmarks like STREAM and PCHASE, as they cause severe contention on shared resources in memory hierarchy.

There is a trade-off between the amounts of idle cycles to harvest vs. the impact on simulation. Such tradeoff can be managed by tuning the parameters of scheduling

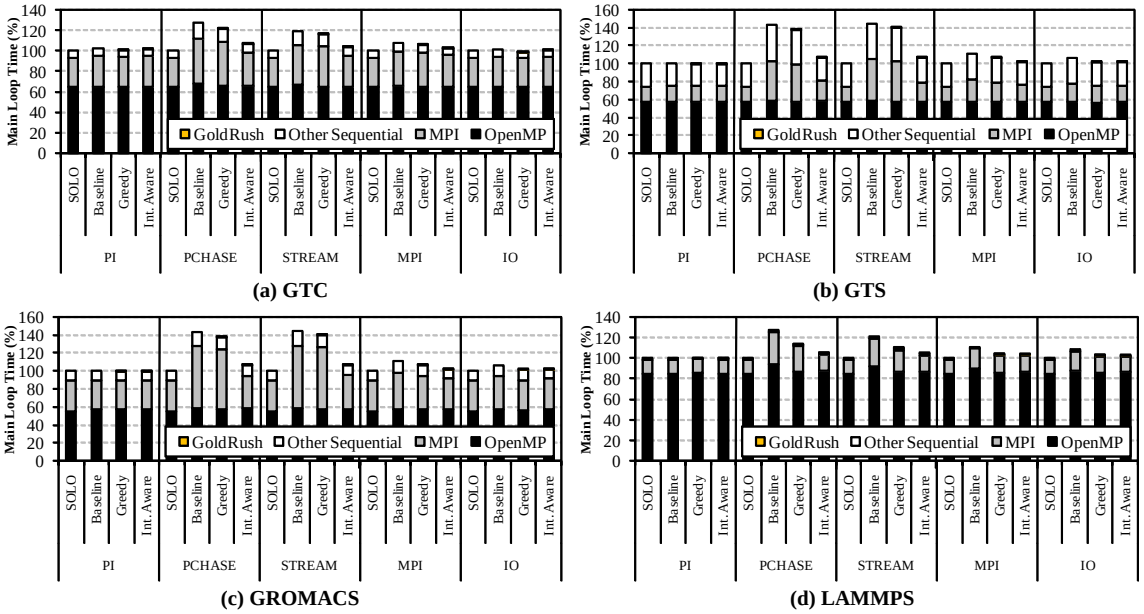


Figure 36: Simulation performance with 1024 cores on Smoky cluster. The legend "GoldRush" refers to the time which the simulation spends in GoldRush operations (monitoring, prediction and signaling). Such overheads are very low (less than 0.3%).

policy (Section 5.3.5). In our tests, we conservatively set the idle period duration selection threshold to 1ms, scheduling interval to 1ms, IPC threshold to 1, L2 Miss Rate to 5, and sleep duration to 200 μs. Such setup results in significant performance improvements at simulation side as shown in Figure 36, and meanwhile the aggregated amount of harvested idle periods is at least 34%, and 64% on average, of total available idle time. A thorough study of parameter tuning is left for future work.

5.4.1.2 Costs of GoldRush

The runtime cost of GoldRush at the simulation side can be quantified by the performance difference between GTS running in solo vs. co-running with analytics under the control of GoldRush. As mentioned earlier, this difference is 1.7% on average.

The simulation side cost of GoldRush can be further divided into two parts: the first part is the time spent in the GoldRush runtime itself (i.e., the time to execute the GoldRush marker APIs and monitoring signal handler), and the second part is

simulation slowdown due to context switches and remaining interference from analytics. We internally instrument GoldRush and find that the aggregated time of the GoldRush runtime itself is small, constituting no more than 0.3% of the simulation's main loop time. Concerning runtime monitoring, the measured memory usage of storing GoldRush monitoring data in main memory is no more than 5KB per simulation process in all test cases.

5.4.2 GTS Application with Online Analytics

GTS (Gyrokinetic Tokamak Simulation) is a global three-dimensional Particle-In-Cell (PIC) code used to study the micro-turbulence and associated transport in magnetically confined fusion plasma of tokamak toroidal devices [139]. GTS outputs particle data during simulation. We apply GoldRush to manage GTS to co-run with two representative particle data analytics.

5.4.2.1 Parallel Coordinate Visual Analytics

Parallel coordinates is a visualization method commonly used to depict and analyze multivariate data [64, 121]. We implement this method for GTS particle data. Each GTS particle has seven attribute, including coordinates, velocities, weight and particle ID. Each processor first generates its local plot of parallel coordinates from the selected particles. Then, all processors collectively generate the final plot through parallel image composition [153]. Multiple plots of parallel coordinates can be generated and composited to show the relationship between different groups of particles. Figure 37 shows the parallel coordinates for two time steps, where the green areas correspond to all particles, and the red areas corresponds to the particles with the absolute 20% largest weights. Our parallel coordinate analytics can clearly show the evolution of particle data distribution at large scale.

GTS is run with a typical setup, which results in particle data output size of 230MB per process. GTS outputs particle data every 20 iterations. Each GTS MPI

process with 6 OpenMP threads is placed onto a separate socket on Hopper's 4-socket compute node. Weak scaling is applied to GTS from 768 to 12288 cores. Within each node, 20 visual analytics processes are placed onto the cores where the simulation's OpenMP threads are running. The 20 analytics processes are divided into 5 groups. Each group has 4 processes with one process running on a separate socket. GTS particles output data of successive timesteps are distributed among the 5 analytics process groups in a round-robin manner via the FlexIO shared memory transport. Both the original particle data and the generated images are written to the file system.

For comparison, we also run GTS and visual analytics "Inline": the simulation directly calls the visual analytics routine. In this way, simulation and analytics are performed synchronously. We use a multi-threaded OpenMP version of the parallel coordinates processing routine to get the best possible inline performance.

Performance: Figure 37 (a) shows the main loop time of GTS simulation with 12288 cores on Hopper. Similar to previous experiments, the performance of GTS is best with GoldRush interference-aware scheduling. "Inline" has worst performance, due to synchronously performing analytics and file I/O. Figure 38 (a) shows the scaling of simulation-side slowdown. The GoldRush interference aware policy has better scalability than the OS baseline solution, which promises its utility at even larger scales.

Cost I (CPU Hours): with the same number of compute nodes used, using GoldRush leads to the least usage of CPU Hours.

Cost II (Data Movement Volumes): an alternative to co-locating simulation and analytics is to perform analytics "In-Transit": additional compute nodes are allocated to host analytics; data is moved from the simulation to analytics through the RDMA-based data staging transport in FlexIO [162]. This makes it possible to avoid contention on compute nodes, but results in additional data movement across

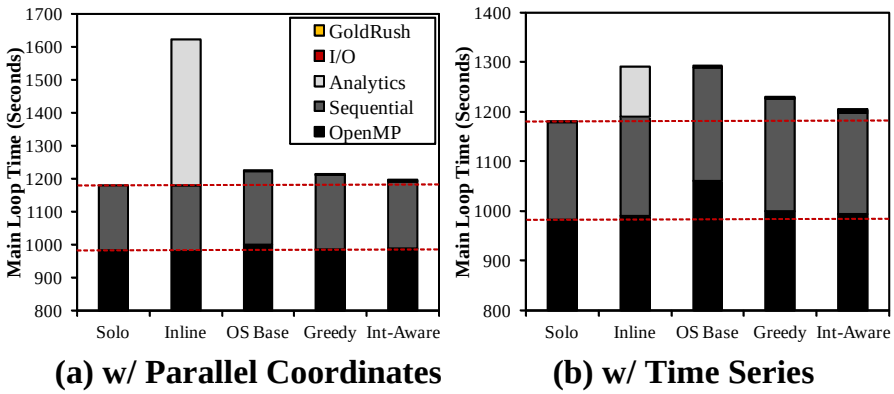


Figure 37: GTS performance with 12288 cores on Hopper.

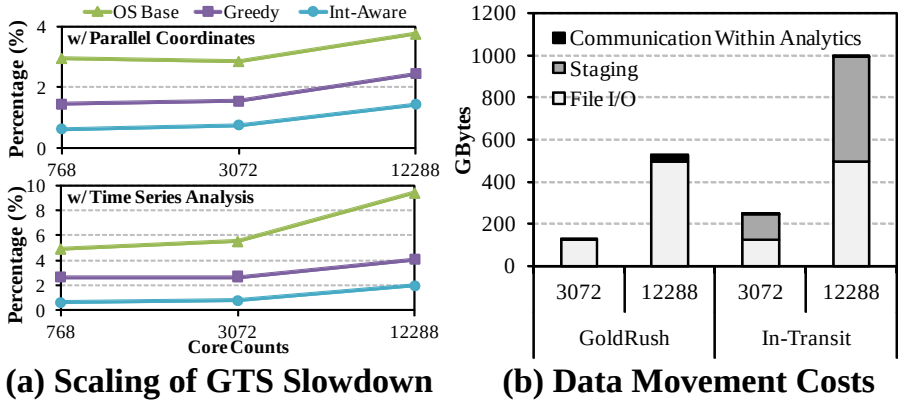


Figure 38: Scaling results on Hopper. Figure (a) shows the slowdown of GTS (comparing to Solo case) with different scheduling policies. Figure (b) compares the data movement costs of running parallel coordinates in situ vs. in transit.

the interconnect (which can also introduce perturbation to simulation [12]). Figure 38 (b) compares the data movement volumes under the GoldRush vs. In-Transit setups, where a 1:128 ratio of compute to staging nodes is used. We note that placing analytics onto a smaller number of staging nodes reduces MPI communication cost within the parallel coordinates analytics (for the image composition), but doing so adds data movements between the simulation and analytics (i.e., the staging traffic). Since placing analytics within the compute node and using GoldRush to schedule its execution can already achieve close-to-optimal performance, it is more efficient to use GoldRush rather than In-Transit for this GTS analytics use case. More generally, of course, In-Transit solutions remain important, because one must "size" on-compute-node analytics to match available idle resources. We leave the creation of general methods for such sizing to future work.

5.4.2.2 Time Series Analytics

Time series analysis [111] is essential for understanding particle temporal behavior. A basic operation of time series analysis is to iteratively access the data of each particle in the arrays of different time steps. A common data access pattern can be simply represented as A[ti][p] = f(B[ti][p], B[ti+1][p]) for two time steps, where, for a particle p, A is a derived variable whose value at the time step ti depends on the original variable B at ti ad ti+1. For example, the displacement of a particle is computed from its positions at two time steps. However, we note that it is non-trivial to generate the particle trajectories in parallel [111, 117], which is out of the scope of this thesis. In our study, we assume that we already have the time-series data of each particle and emulate the data access pattern with a synthetic code.

We co-run the code that exercises the data access pattern on GTS particle output data. Due to its streaming access pattern, the time series analytics causes 15.2 L2 cache misses per thousand instructions on Hopper. As shown in Figure 37 (b) and

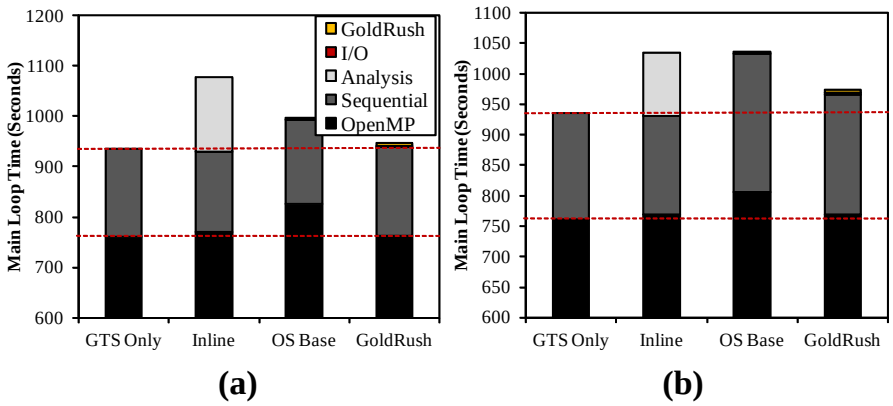


Figure 39: Simulation and analytics execution time on Intel Westmere machine.

38 (a), this results in up to 9.4% slowdown of the GTS simulation with 12288 cores under the OS scheduler. The GoldRush interference aware scheduler reduces such interference to at most 1.9% and manages to complete all analytics processing with available idle resources.

5.4.3 Varying Architecture - Intel Westmere

In order to evaluate the effectiveness of GoldRush across different architectures and its scalability within a node, we conduct experiments on a 32-core Intel Westmere machine. The machine has 4 sockets each with 8 cores at 2.13GHz, with a 32KB D-Cache, 32KB I-Cache, and a 256KB inclusive unified L2 Cache. All 8 cores within a socket share a 24MB inclusive unified L3 Cache. Each of the 4 sockets belongs to one NUMA memory domain with 32GB DDR3 memory in each domain. We run GTS with 4 MPI processes and 8 threads per process on this machine.

Figure 39 (a) shows the main loop time of the GTS simulation co-running with parallel coordinate analytics. The simulation's OpenMP time increases by up to 5% under OS scheduler. This is because the OS scheduler does not entirely suspend analytics and thus, incurs unnecessary scheduling overhead. GoldRush with its greedy policy, however, results in GTS performance within 99% of the optimal. The less than 1% performance loss is due to time spent in the shared memory transport and the

GoldRush runtime.

When co-running GTS with the contentious time series analytics under OS baseline scheduling, GTS can be significantly slowed down (up to 11%), as shown in Figure 39 (b). On the other hand, with the interference aware GoldRush scheduling, interference is again greatly reduced. This, together with previous results, demonstrates GoldRush's ability to mitigate interference between co-running simulation and analytics across different architectures.

5.5 Related Work

In Situ Scientific Data Analytics. In situ data analytics and visualization has gained much recent attention from the HPC community. Current work falls into two categories: (i) data analytics and visualization algorithms, and (ii) supporting platforms. Regarding the first category, the GoldRush system can be readily used to run various data analytics with idle resources in compute nodes for resource-efficient, near-source data processing. As to the second category, systems like PreDatA, GLEAN [94], Nessie [61] and DataSpaces [1], all support In-Transit data processing (i.e., deploy data analytics on auxiliary nodes and move data from simulation to analytics across interconnect), which is orthogonal to our work. One attractive usage of GoldRush is to run data-reducing operations on compute nodes to filter or pre-process simulation output data before sending data to In-Transit processing nodes. Finally, Damaris [43] and Functional Partitioning [81] use dedicated cores on compute nodes for file I/O and other data operations; such solutions are easily realized with GoldRush.

Cycle Stealing. Idle CPU cycles pervasively exist on PCs and servers. There has been extensive work on leveraging unused idle cycles for useful computation. Examples include Condor [82], BOINC [24], and other volunteer computing systems. To the best of our knowledge, GoldRush is the first system to harvest fine-grained idle

cycles from large-scale scientific simulations on HEC platforms for online data analytics. Linger Longer [122] shares similarity with our work since it enables aggressive resource sharing between host applications and background jobs. GoldRush differs in its demonstrated scalability and in its ability to control interference for tightly synchronized host applications (parallel simulations).

Contention Mitigation on Multi-Core Platforms. Resource contention has been recognized as a severe performance issue for consolidated workloads on multi-core platforms. Software solutions to this problem include thread mapping [163, 161] and scheduling [79], cache partitioning [92], and compiler-time code transformation for cache behavior optimization [123, 134]. We borrow from such work to implement a special case for contention aware scheduling in which analytics processes detect contention with high-priority simulation and dynamically back off to mitigate interference. Most similar to our work are CAER [96] and ReQoS [135], both of which target data center applications.

Optimizing MPI/OpenMP Hybrid Programs. There has been previous work on tuning performance and/or power efficiency for MPI/OpenMP hybrid codes. One approach is to overlap sequential code regions with parallel OpenMP regions [118], but its applicability is application-specific, constrained by data and control dependencies as well as by thread safety in MPI libraries. Another approach by Li et al. [80] applies Dynamic Concurrency Throttling and Dynamic Voltage and Frequency Scaling to OpenMP parallel phases, for power savings. The slack prediction used in that approach is akin to GoldRush's idle period prediction: it estimates the difference in the duration of OpenMP parallel phases between the non-critical vs. critical (the slowest) MPI processes (i.e., the slack time), so that the non-critical processes can be run with reduced CPU frequencies during slack time. The duration of a sequential period between two successive OpenMP parallel phases is measured directly and used as input parameters to slack prediction. GoldRush, instead, dynamically predicts

the duration of those sequential periods within each MPI process; and its purpose is not to reduce the power consumption of a simulation code running in solo, but to orchestrate the execution of coupled simulation and analytics to improve their overall performance and resource efficiency. It would be interesting, however, for a MPI/OpenMP hybrid simulation code, to use both methods: to optimize power efficiency of the OpenMP parallel phases and to apply GoldRush to schedule analytics during idle periods outside the OpenMP phases. Also interesting to GoldRush is to leverage Li's work on dynamically varying OpenMP thread count for simulation's OpenMP phases: this may help yield even more idle resources for online analytics.

5.6 Conclusions

This chapter makes several key contributions to improving the online execution of data analytics. We first show that even leadership simulations leave considerable compute node resources unused. This is not because such codes are ill-tuned or configured, but because many such unused resources often occur as modestly sized idle periods not easily utilized by the dense core methods constituting the bulk of a typical simulation's computation. Unfortunately, this fact also makes such idle periods difficult to use for analytics. This is the key challenge addressed by the GoldRush system developed in our work. GoldRush applies fine-grained scheduling to "steal" idle resources from simulation in ways that incur negligible runtime overheads and minimize interference between the simulation and analytics. Key to its effectiveness are (i) judiciously selecting appropriate idle periods based on online performance monitoring and prediction, and (ii) dynamically detecting interference and mitigating it by throttling analytics execution. Experiments with representative applications at large scales (up to 12288 cores) and on different architectures show that resources harvested on compute nodes can be used to perform useful analytics, significantly improving resource efficiency, reducing data movement costs incurred by alternate

solutions, and posing negligible impact on simulations.

CHAPTER VI

SUPPORTING ONLINE SPATIAL INDICES

6.1 Introduction

In Chapter II, we identify rich metadata support as one of the essential requirements by online scientific data analytics for both interoperability and performance enhancement purposes. The work described so far has focused on how to use the rich metadata requirements to develop more efficient in situ and staging computation layouts based on a user's end-to-end intent. However, given the evolution of hardware towards deep memory hierarchies, it is important to understand the capabilities that knowledge of user intent can enable in memory placement and movement.

Among the various types of meta-data organizations, spatial indices represent an important case due to their wide usage and applicability to scientific data. Correspondingly, we have chosen them as an exemplar of a user-driven organizational requirement with very strong memory placement issues in deep memory hierarchies. A spatial index partitions the attribute domain into a bounded logical hierarchy that allows for quick responses to region-based queries in massive data sets. Spatial indices are used by many applications to accelerate analysis and visualizations, since many instances of both require access to subsets of data that are near to particular marker points. Popular spatial indices include R-Trees, Octrees, and KD-Trees.

In order to use spatial indices for online scientific data analytics, however, significant challenges need to be addressed to achieve scalability and memory efficiency. The spatial indices need to be constructed from live simulation output data in an online/in-memory fashion. Additionally, analytics functions dependent upon tree-based range queries might require regular updates during the bulk construction of

the tree. The overheads of generating, buffering and (possibly) transferring spatial indices therefore need to be carefully managed and tuned to avoid overwhelming the overall I/O pipeline or per-node memory subsystem.

Further, most analytics run as parallel programs, and their performance can be highly sensitive to misjudged data distributions. This in turn implies that the distribution of both the spatial indices and the indexed data need to be carefully load balanced with awareness of user's queries, even in the face of skewed data distributions. Although there exists extensive prior work on distributed spatial indices in the context of database, GIS, and pure visualization research, to date little work has been done in the online and in situ construction, distribution, and querying of online spatial indices for large-scale, streaming scientific simulation output data.

ItThis problem has become increasingly important due to the challenges inherent within architectural trends towards deeper memory hierarchy on High End Computing machines. The rapid increase of total data size as we approach the Exascale is pushing data-intensive scientific applications to the DRAM capacity wall, and DRAM's high power consumption (up to 40% of total machine power usage) also poses great pressure on the tight energy budget of current generation and future Exascale machines. These two trends promote the use of Non-Volatile Memory (NVM) such as Flash Solid State Drives (SSD) to extend DRAM capacity, resulting in a deep memory hierarchy spanning from CPU cache, DRAM, Flash SSD, to hard-drive disks even on today's hardware. Looking forward, the types and architectural hierarchies of memory will only become more varied.

Such deep memory hierarchies make indexing of large scientific arrays increasingly important. By the sheer size (10's to 100's of terabytes per data output) of the scientific data, it is guaranteed it will need to inhabit different tiers and different parallel locations within leadership class machines. Due to the ability to filter out accesses to irrelevant data as well as improving the performance of direct queries, such

built-in indexing schemes can dramatically impact memory efficiency. On the other hand, constructing and querying spatial index on the deep memory hierarchy requires non-trivial optimization on index structure and data movement between different levels of the hierarchies, due to the differing natures of the underlying hardware. As an example, SSDs have distinct I/O characteristics such as read/write asymmetry and limited erase cycles which have been shown to render existing OS paging and database's buffering policies sub-optimal, demonstrating a need for special treatment in optimizing the data movement between memory tiers.

In this chapter, we propose a general framework called ZStore which enables the construction of spatial indices from live simulation output data and the application of those indices to speed up commonly used queries and visualization tasks. ZStore is a middleware-level service that provides a customization interface to incorporate application-specific data distribution policies. These policies allow ZStore to respect application intents with respect to the consumption of the massive I/O data streams. To demonstrate its general applicability, we use ZStore to implement two representative spatial indices: RTree and Octree indices. Our use cases also demonstrate their use with several common types of queries associated with the two indices. We show that both RTree and Octree can be used to index live simulation output data and answer online queries, and they can achieve load balance and high performance at large scales.

We apply ZStore to two real-world scientific applications, S3D and LAMMPS, and show the spatial indices built with ZStore can significantly improve the end-to-end performance of the overall I/O pipelines for both applications. Experimental results also show that for both applications, the analytics implemented as out-of-core programs using ZStore can achieve comparable performance with their in-core versions while utilizing significantly less DRAM, and hence opening up new ways of balancing both power and memory allocations within multi-component I/O workflows. This

demonstrates that ZStore can be a valuable software solution to overcome the DRAM capacity wall.

The reminder of this chapter is organized as follows. Section 6.2 demonstrates the technical challenges of supporting spatial index in the context of online data analytics and deep memory hierarchy. Section 6.3 describes the design of ZStore and its key components. Section 6.4 describes how to implement and optimize RTree and Octree indices using ZStore. Section 6.5 describes the usage and sample applications of ZStore. Section 6.6 provides performance evaluation results. Section 6.7 reviews related work, and Section 6.8 draws conclusions.

6.2 Background and Motivation

6.2.1 Spatial Indices for Online Scientific Data Analytics

Below, we briefly summarize some of the cogent points about the two spatial trees that we have chosen to examine in this work: RTrees and Octrees. They both represent popular and useful approaches for recursively generating spatial indices [56, 97].

1) RTree Index. The RTree [55] was designed to index multi-dimensional data sets. There are many variants, but conceptually they all use the same basic approach, which is to recursively divide space into a set of possibly overlapping rectangles. The details come down to how such divisions are achieved and rebalanced when individual data points arrive and depart regularly.

More exactly, in an RTree each node can be either a point or a collection of non-zero sized polyhedra, and it is associated with a Minimum Bounding Rectangle (MBR). Each Rtree node can contain at most B entries, and with the exception of the root node, at least $b \geq B$ entries. Each leaf node entry contains a MBR and the data object (or reference to a data object stored externally). Each internal node entry contains the pointer to a child node as well as a MBR which is the minimum bounding rectangle fully containing all downward data objects. All leaf nodes in an RTree are

at the same level, so an RTree is height-balanced and its height is $O(log_b n)$, where n is the number of data objects being indexed. Note that the MBRs of entries at the same level can overlap, and different RTrees may be constructed from the same set of input data objects by varying the insertion order. Many variants like R+Tree and R*Tree have been devised to reduce chances of node splits during insertion and/or reduce the overlap among entries to improve query performance.

2) Octree Index. An Octree [97] usually refers to a class of hierarchical spatial tree structures: Quad-tree for 2-dimensional space, Octree for 3-dimensional space, and Hyperoctree for dimensions higher than three. We will focus our discussion on the 3-dimensional Octree as an example, since it is extremely prevalent in some types of scientific analysis work. The Octree index recursively partitions the 3D space into 8 sub-regions (also called octants or "Cells") using separators parallel to the coordinate axis. The spatial partition stops at a pre-defined granularity or condition (for example, when the volume of the child cell is less than some characteristic length of the simulation). Each internal node of an Octree has at most 8 entries, each corresponding to one of the 8 Cells at that level of partitioning granularity. A leaf node in an Octree only has one entry and corresponds to one Cell of corresponding granularity. Octrees need not be height-balanced: the recursive partitioning may stop at a node as long as certain pre-specified condition is met; for example, the difference between minimum and maximum values recorded within the volume is less than some target error. This makes Octree very useful to represent multi-resolution data sets: regions of interest can be partitioned at finer granularity to gain deeper level of detail, while un-interesting or slowly varying regions can be quickly summarized without further lookup or storage cost. Another interesting property of Octree is that the Cells derived from Octree partitioning can be identified and ordered by space-filling curves such as Z-curve and Hilbert curve.

Despite their structural differences, both RTree and Octree have shown to be able

to significantly accelerate various spatial queries. As we mentioned earlier in Chapter II, applications like LAMMPS and S3D can greatly benefit from using RTree and Octree in their online data analytics and visualization tasks. This motivates us to enable online spatial indexing.

6.2.2 SSD-Equipped Deep Memory Hierarchy

As mentioned in Introduction section, DRAM has become an expensive and scarce resource on High End Computing platforms. DRAM is both expensive in price and power compared to many of the other individual components in current machines. Projecting trends forward, one can see that the amount of DRAM available per core is dropping significantly, as memory bandwidths and sizes stagnate in the comparison to the rate of increasing core count. This "DRAM capacity wall" has severe impact on data-intensive workloads in general and on online scientific data analytics in particular. Existing online data analytics solutions (including the approaches we have proposed in previous Chapters) commonly buffer simulation output data in DRAM for analytics processing, which therefore is constrained by the available DRAM. Although the GoldRush approach can harvest idle DRAM allocations from the scientific simulation in order to improve memory usage efficiency, the amount of data which can be buffered and processed is still limited by DRAM capacity. For cases where the simulation is memory hungry and/or the simulation output data size is too large to be buffered with the remaining DRAM space, the runtime for generating efficient in situ management will need to be adapted to focus more on memory placement than on computation.

Realizing that the current HEC architecture may poorly match current and future data-intensive applications, researchers have proposed to extend DRAM with Non-Volatile Memory to overcome the DRAM capacity wall. Among alternative solutions, Flash-based Solid State Drives (SSDs) have shown promise in the short term

as a way of extending the memory hierarchy of production environments. In comparison to Hard-Disk Drives (HDDs), SSD has many advantages such as high random I/O performance, higher density, lower power consumption, stronger shock resistance, etc. Previous work has explored various ways to incorporate SSD into existing architecture: One is to use SSD along with HDD to form a hybrid storage system. Yet another is to install SSD locally on all or a subset of compute nodes. [65] uses both model simulation and hardware prototypes to demonstrate the latter approach (i.e., node-local SSD). It shows that it is possible to bring SSDs closer to CPUs in the hierarchy while having performance and cost advantages over the former approach. This work leverages such previous scholarship to form our hardware base case, and we will specifically be investigating scenarios where staging activities might take place on potentially specialized nodes with both DRAM and SSDs, in part as a stand-in for future PCM or other non-volatile memory technologies.

6.2.3 Technical Challenges

Our goal is to demonstrate that the construction of a user-specified spatial index from live simulation output data and its use for online analytics and visualization can be accelerated through careful distributed data and computation placement. We are also inspired by the trend of deep memory hierarchy and would like to investigate the trade-offs and capabilities available when one has access to node-local SSDs for online and in situ analytics. Towards these goals, the following challenges must be addressed:

1) **Constructing Spatial Index on Distributed Data Streams**. The spatial index needs to be constructed from live simulation output data streams in an online fashion and made available to the user-defined analytics process by answering queries. The computation and storage costs of constructing, buffering, transferring, and assembling the index should be minimized to avoid overwhelming the overall I/O

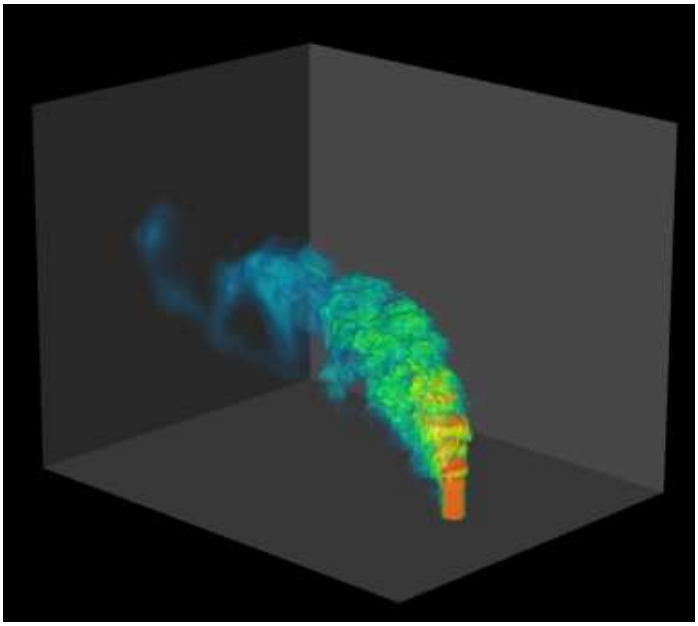


Figure 40: Sample S3D volume data.

pipeline.

2) **Achieving Load Balance with Distributed Index**. Most leadership-class analytics run as parallel programs, and their performance is sensitive to data distribution and can suffer severely from unbalanced data distribution. As an example, consider the S3D data in Figure 40. The feature of interest (the frame front) is distributed in a highly skewed fashion within the simulation volume data. Consequently, an Octree spatial index and leaf data, which partitions the space rather than the elements of interest, would require careful distribution in order to achieve parallel load balance for the parallel analytics.

3) **Achieving High Performance and Memory Efficient Indexing on Deep Memory Hierarchy**. SSDs have unique I/O characteristics (such as read/write asymmetry and limited erase cycles) which must be considered to optimize any manual paging of scientific data between DRAM and SSD. A key goal is to generalize the resource co-scheduling and placement techniques visited in earlier work so that the price of utilizing the different classes of memory can be minimized. In order

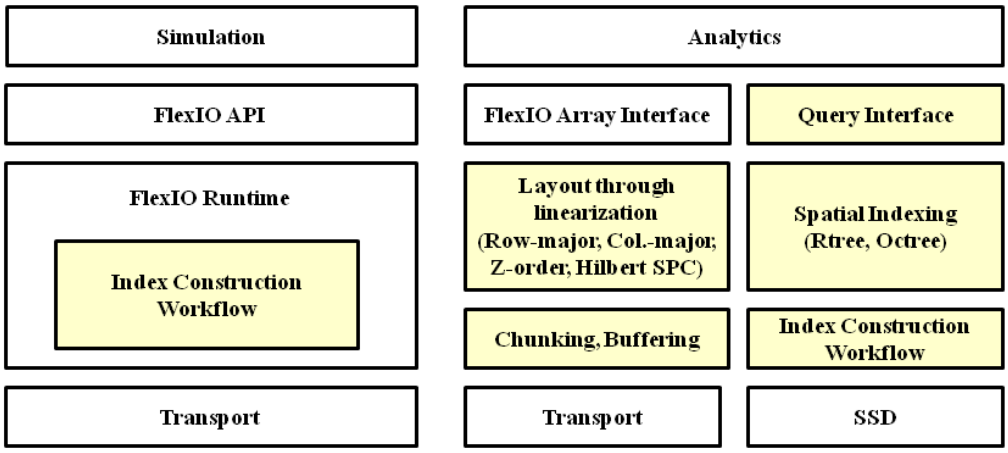


Figure 41: The architecture of ZStore.

to achieve performance comparable to an in-core counterpart, the runtime will be required to enhance data locality and reduce the effect of data movement latencies between different levels of the hierarchies.

6.3 ZStore Design and Implementation

6.3.1 Overview

ZStore is the distributed framework for testing and evaluating the impact of different control strategies for indexing and querying multi-dimensional array data on compute nodes equipped with SSD-extended memory hierarchies. Specifically, it provides necessary middleware-level services required to implement spatial indices such as RTree and Octree, and it also allows end users to exploit this indexed simulation output data for in situ or online analytics queries.

As shown in Figure 41, ZStore consists of three major components: i) a flexible in-transit index construction workflow; ii) a buffer manager for efficient data movement between DRAM and SSD; and iii) an multi-dimensional array store.

ZStore leverages a flexible in-transit workflow descended from the two-pass streaming processing model described in the earlier PreDatA work in Chapter III. This workflow decomposes the index construction process into separate stages which can

then be distributed along the compute resources of the I/O path in order to achieve scalability.

In the previous work, we have seen the importance of the ability to embed user-specified code directly into the course of data movement. Building on that recognition, ZStore provides a customization interface to incorporate application-specific data distribution policies. The data distribution policy is implemented as two callback functions which are invoked as part of the index construction workflow. These functions then allow for user- or domain-specific partitioning and chunking of data. This design gives application flexibility in controlling data distribution, as the optimal data distribution is always specific to the target application. For ease of use, ZStore also provides a set of built-in data distribution polices for common cases, and a user can choose one of them as a starting point to further customization and control.

To address the challenge of constructing and querying spatial index on deep memory hierarchy, ZStore jointly applies a set of data layout and buffering optimization techniques. These allow us to both improve the particular effectiveness of the stores we will later demonstrate, but also to explore the nature of optimizations relevant for future hierarchies of computation and storage. In particular, ZStore adopts a packed column-oriented data layout which not only improves cache efficiency, but it also enable the use of SIMD vector instructions to speed up the critical computation routines. ZStore also provides a common buffer manager service to handle the data movement between DRAM and SSD, and it uses a buffer replacement policy which prioritizes clean pages over dirty pages to respect the read/write asymmetry of SSDs.

In addition to the index generation capabilities for analytics, ZStore provides built-in storage methods for storing and accessing scientific muti-dimensional array data for SSD-based, out-of-core use. This native data storage layer supports various commonly used array layouts, including row-major, column-major, Z-order (Morton layout), and Hilbert space-filling curve.

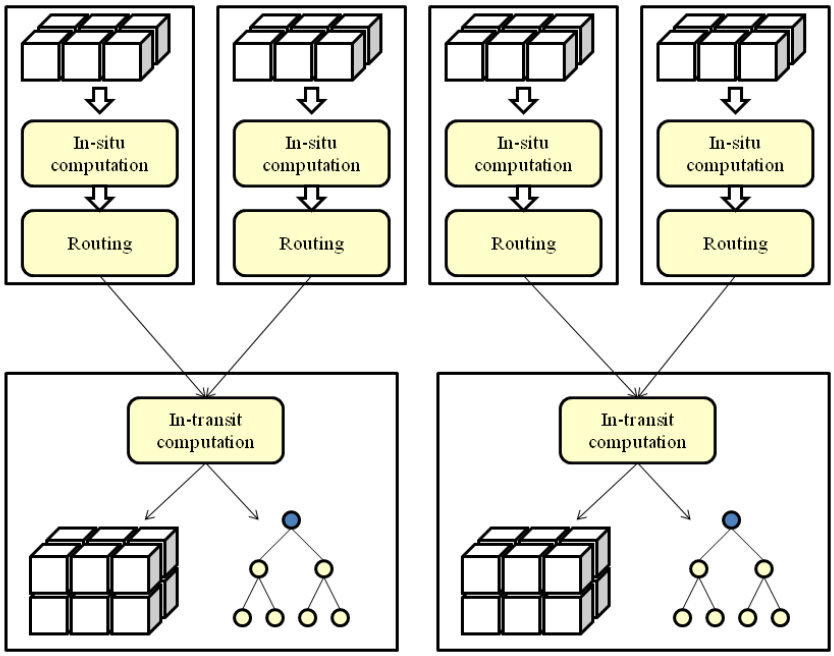


Figure 42: ZStore Index Construction Workflow

6.3.2 Constructing Spatial Index on Distributed Data Streams

ZStore provides a flexible in-transit index construction workflow to build and distribute index from live simulation output data streams. The workflow decomposes the index construction into multiple stages, and deploys these stages at appropriate places along the I/O path. The workflow is triggered and executed while simulation output data is moved from simulation to analytics, and the generated spatial index is transferred (along with the original simulation output), distributed, and finally made available to analytics for answering queries. To construct a specific spatial index online, each stage of the workflow is implemented and registered as a callback function (much like the MapReduce programming model and the PreDatA model described in Chapter II), and invoked by ZStore automatically at runtime. This in-transit workflow is embedded inside the FlexIO I/O middleware, and is transparent to the simulation code.

Figure 42 shows the overall index construction workflow. When each simulation

process has data to output, it passes data and control to the FlexIO runtime. Inside FlexIO, each simulation process executes an optional workflow stage termed "*inline computation*" implemented by a callback function inline_compute() if present. This stage takes the local simulation output data as input, and can perform certain computation and possibly generate additional data (e.g., generate a local index for the local portion of a global array). Within the insitu_compute() function, one can call a callback function chunking() to break the original simulation output data into chunks. If the chunking() function is not provided, then the default is to treat each simulation output array as a chunk. The inline_compute() function is also provided with a MPI communicator as a parameter which contains all simulation processes participating in the I/O action, so MPI collective communication can be used within the function if necessary (e.g., for calculating global min/max or determining skews in global-level data distribution). The output data of the *inline computation* stage is then transferred along with the original simulation output data chunks to the analytics side. For each chunk, a routing() function is called to determines which analytics process to send this chunk to. The routing () function can be implemented by application to control data distribution at analytics side. There are also a set of built-in routing() functions each of which implements a common distribution policy (More on this topic in Section 6.3.3).

The analytics processes receive simulation output data in chunks, along with the associated data additionally generated by the in-transit workflow. When a chunk and its associated data is received and before returning to analytics, a workflow stage called "*in-transit computation*" and implemented by a callback function in-transit_compute() is executed. This stage takes as input the simulation output data chunk and the associated data which is generated by previous stages of the workflow. Typically, the "*in-transit computation*" stage merges the local indices into a global index which is stored in an internal storage container (can be either in-memory or

out-of-core). When all chunks are received, a *finalize* stage is executed to perform any necessary cleanup tasks. After the *finalize* stage completes, the simulation output data and constructed index is made available for analytics.

This index construction workflow is sufficiently flexible to implement a variety of spatial indices (we will show how to implement RTree and Octree using this workflow later). Note the similarity between this workflow and the PreDatA streaming processing model in Chapter II. One can think of this index construction workflow as a specialized instance of PreDatA.

Also note that at the analytics side, the original simulation output data and generated spatial index are stored in a storage container. If the node is equipped with SSD, the container can be configured as a DRAM buffer coupled with backend files on SSD, to allow out-of-core array and index accesses. In this case, a buffer manager automatically manages the DRAM buffer space and the I/O between DRAM buffer and backend files. We will describe the buffer manager in Section 6.3.5.

6.3.3 Controlling Data Distribution at Analytics Side

For distributed index and query processing, the distribution of both the index and original data among analytics processes can have significant impact on query performance. The ideal distribution should achieve load balance and minimize remote data accesses. Controlling data distribution for parallel processing is a well studied topic in both database and parallel computing communities. The basic lesson is that the optimal data distribution is always specific to the data itself and the query workloads. In our experiences, we find that scientists usually either have a good knowledge of how to partition the data, or already use certain existing data partitioning tool such as METIS to determine the appropriate data distribution. Therefore, regarding balanced data distribution, we do not claim any contribution in novel data distribution and partitioning algorithms. Instead, our ZStore framework gives control over data

distribution to the application. This is realized via the routing() and the chunking() callback functions in the index construction workflow which allows application-specific policy to decide how simulation output data and associated spatial index is distributed among the analytics processes. Later we will show non-trivial application use cases to demonstrate the usage of this feature.

ZStore also has a set of built-in data distribution policies which cover a range of common scenarios and can be easily configured to use. These built-in policies are all implemented as routing() callback functions in the same way as user-defined routing() functions. The supported built-in policies include the following: i) round-robin distribution; ii) randomized distribution; and iii) block-block distribution.

The routing() function enforces data distribution during data movement from simulation to analytics. Once data arrives at the analytics side, no further data re-distribution will be performed. One potential extension to this is to enable data re-distribution so that the initial distribution can be dynamically adjusted (e.g., based on observed load balance of queries). However, the necessity and benefit of such re-distribution is not clear yet from our application experiences and we leave this topic as part of our future work.

6.3.4 Query with Distributed Spatial Index

Once simulation output data and associated spatial index are available at analytics side, the analytics processes can perform various queries on data by using the index. Each specific spatial index built with ZStore needs to implement its own query routines. ZStore, however, provides common utilities to facilitate the query implementation. These mainly include a buffer manager for out-of-core index (described in Section 6.3.5) and a meta-data replication utility which is described below.

To enable query on distributed index and data arrays, ZStore allows replicating spatial index and meta-data regarding array distribution among multiple analytics

processes. For a distributed array, its distribution information describes which portions of this array resides on which analytics processes. Such information is actually generated in FlexIO's data transfer protocol (see Chapter IV), and broadcasted among all analytics processes. Therefore, each analytics process has a replication of a global array's distribution information and can locate any specific region of the global array (in terms of which analytics process owns it).

For a spatial index constructed from a distributed global array, the index implementation can selectively replicate a portion of the global index among all analytics processes. The most common case is to replicate the top few levels of the index tree among all analytics processes. Nodes at the lowest level of the replication contain two types of entries: a local entry point to a local sub-index or data, and a remote entry specifies the region residing on some remote process and the ID of that remote process. The remote lookup is internally implemented using MPI, and remote lookup results are cached in a fix-sized local memory buffer using simple LRU eviction policy.

With the replicated distributed index, query on a distribute data set can be executed in parallel. A typical execution plan is for each analytics process to query its local index and data, and then leverage the replicated global index to perform remote lookup, and finally assemble and/or aggregate the query results. In Section 6.4, we will show examples of distributed queries which are implemented with index replication.

6.3.5 Buffer Manager for Out-of-Core Index on SSD

With the presence of SSD-equipped deep memory hierarchy, ZStore can support out-of-core index and multi-dimensional arrays which are larger than DRAM capacity on a single node. A key component for this is a buffer manager which manages a DRAM buffer and handles I/O automatically between the DRAM buffer and backend files on SSD.

In ZStore, any out-of-core data (either the spatial index or data array) is implemented via a fix-sized DRAM buffer backed by file(s) on the SSD. A DRAM buffer is of fixed total size, and contains a number of fix-sized blocks. The data resides in the backend file(s) on SSD and is brought into or written from the DRAM buffer during read/write accesses. The I/O unit between the DRAM buffer and SSD backend is a block. The DRAM buffer is highly flexible, and its total size, block size, and buffering policies can all be configured to tune performance. ZStore's buffer manager essentially enables "application-managed paging" at user level. It is more flexible and efficient than OS paging, and more transparent and portable than explicit file I/O.

ZStore's buffer manager features a SSD-optimized buffer replacement policy. SSD has two important I/O characteristics: i) read/write asymmetry, and ii) limited program/erase cycles. Read/write asymmetry implies that since writing to the same location in SSD requires erase before program the Flash cells, writing is notably slower than reading for SSD. Besides, each Flash cell can be only overwritten for a limited number of times which then set a limited life time for the SSD. This has performance implication on buffer replacement policy: when selecting a victim buffer block to evict to make room for another block in DRAM buffer, evicting a dirty block is slower than evicting a clean block, since replacing a dirty block requires writing the block back to backend file on SSD, which takes more time than replacing a clean page.

ZStore leverages the second-chance Clock algorithm to address the read-write asymmetry issue with SSD. This buffer replacement policy prioritizes clean buffer blocks over dirty blocks for replacement. As a result, the write performance penalty is mitigated and meanwhile temporal locality is approximated.

6.3.6 Storing Multi-Dimensional Arrays on SSD

Besides out-of-core spatial indices, ZStore also supports storing and accessing out-of-core multi-dimensional arrays on SSD. A multi-dimensional array is stored in a back-end file on local SSD. Analytics programs access the array via a high level read/write interface which specifies the array indices of array portion to access. Internally, a DRAM buffer is used to automatically handle the data movement between DRAM and SSD. Like the buffering for out-of-core index, the DRAM buffer for an array is of fixed size and consists of fix-sized blocks, and various replacement policies can be chosen to manage the buffer in a way suitable for the analytics' access pattern.

ZStore separates the array index from the array's internal layout, and supports four commonly used internal layouts: i) row-major, ii) column-major, iii) Z-order (Morton layout), and iv) Hilbert space filling curve. All the four types of layout can be chosen from high-level programming interface and enforced by ZStore. ZStore automatically handles the translation from array index to the internal ordering of the chosen layout. This gives analytics the flexibility to optimize array layout to best match its access pattern.

Using ZStore's out-of-core array, each node can hold arrays which are larger than node's DRAM space. This capability, together with ZStore's distributed index and array distribution meta-data, allows application to consolidate multiple nodes' DRAMs and SSDs for analytics on massive datasets.

6.4 Implementing and Optimizing Spatial Indices with ZStore

To demonstrate the flexibility and generality of ZStore, we implement two representative spatial indices, RTree and Octree, with the ZStore framework. Both indices can be generated online from simulation output data by using the in-transit index construction workflow, and used to answer queries at analytics side. For each type

of index, we also propose a set of optimization techniques which achieve notable performance improvements.

6.4.1 RTree Index

Constructing RTree Index. When constructing a RTree index, the *inline computation* stage performed at each simulation process breaks the local array into chunks as directed by the chunking callback function, and calls the routing function to distribute the chunks to downstream analytics processes. At the analytics side, each incoming data chunk is inserted into a local RTree (can be out-of-core if the index is to be stored on local SSD). The insertion is done via a bulk insertion routine (we have currently implemented the STR and seed tree bulk loading and insertion algorithms). In the RTree, each data chunk is an input object and its overall MBR is used as its MBR. Once all the data chunks are received and inserted into the local RTrees, analytics processes replicate the upper K levels of their local RTrees among themselves (K is a user-defined parameter). This completes the index construction workflow. At this point, each analytics process has a local RTree index to cover the local data chunks, and an replicated upper portion of the global Rtree.

Note that each application may customize the chunking and routing functions to achieve different data distributions at analytics side. For example, if the RTree index is used to answer range queries, then the data chunks are better to be distributed using a randomized or random robin policy so that the query load is balanced among analytics processes. On the other hand, if the RTree index is used for nearest neighbor search or spherical search for every data object (as is the case for LAMMPS Bond analytics), then the data chunks should be clustered by their spatial approximation so that remote lookups can be minimized. Besides, the chunk size and the replication level can be tuned to control the size of RTree index.

Querying RTree Index. The RTree index can be used to answer range queries

and various types of spatial queries (including containment, intersection, etc.). It can also be used for nearest neighbor search. Distributed queries with RTree is enabled by using the partially replicated global index: each analytics process uses the upper levels of the global index to determine which peer processes (if any) may contains qualifying data and sends queries to corresponding process(es).

RTree Optimization: Column-oriented Node Layout and SIMD Vectorization. RTree implementations commonly store entries in nodes as an array of entries, and each entry consists of coordinates of a MBR and pointer to child (for internal nodes) or data object (for leaf nodes). During insertion and query, the entries of a node are examined one by one by applying the same arithmetic computations (such as calculating overlapping or area enlargement).

Our implementation, on the contrary, adopts a column-oriented node layout: the same coordinates of all MBRs are stored together and sequentially. Such a column-oriented layout has a great advantage: it enables use of SIMD vectorization instructions to achieve instruction-level data parallelism in many critical computation portions of insertion and query. Besides, maintaining the column-oriented layout does not incur any additional measurable cost in comparison to the conventional array-of-entries layout.

In the Performance Evaluation Section, we will show that the column-oriented layout and SIMD vectorization can accelerate both insertion and query performance by up to 60

6.4.2 Octree Index

Constructing and Querying Octree Index. The Octree index can be constructed similarly as RTree index. There is, however, one notable difference: an Octree can be height-imbalanced, and the Octree can grow to deeper levels to cover the regions of interest at finer granularity. Accordingly, in the *inline computation* stage, each

simulation process can generate chunks at different levels of granularity (each chunk will be covered by an Octree leaf node at a corresponding level).

In our Octree implementation, we use several novel techniques to improve memory efficiency and performance, described below.

Octree Optimization I: Breath-First, Z-Order Layout. We order the nodes in an Octree first by level so that nodes of the same level are stored together. Within the same level, nodes are further ordered by the Z-values of corresponding Cells. Such an ordered layout has three advantages: i) it makes breadth-first search access the index sequentially; ii) it makes it easy to access all nodes at a specific level of the Octree in case user wants to explore data at a certain level of detail; iii) the Z-order preserves spatial locality (nodes stored close to each other also index Cells which are nearby in space).

Octree Optimization II: Compact "Node Set" Representation. By definition, each Octree node's children are derived by partitioning the parent node's Cell into sub-Cells. Among the eight sibling nodes, they have many attributes which are either identical or related to each other. For example, the tree level of all the siblings is the same; the Z-values of all the siblings are successive in their tree level.

We exploit this fact by using a representation of Octree nodes which is much more compact than storing individual nodes. The idea is to store all eight siblings together into a data structure called "Node Set". Within a Node Set, there can be up to eight sibling nodes, and the same attributes of the nodes are stored together. This not only eliminates redundant copies of identical attributes, but also makes it possible to represent relevant attributes more compactly. For example, we only need to store the Z-value of the first node and the other siblings' Z-values can be derived very easily. The "is_leaf" flag, as another example, can be represented by 1 Byte for all 8 siblings; If we represent each node separately, it would require 1 Byte per node and hence 8 Bytes for 8 siblings (in practice it may cost 16 or even 32 bytes if the

compiler packs the data structure for alignment). Furthermore, when using the Node Set representation together with the breadth-first, Z-order layout, we only need to save one pointer which point to the child with the least Z-value among all 8 siblings, since siblings' children are guaranteed to be stored successively in this layout. On the other hand, in the conventional per-node representation, each node would need 1 (for breadth-first, Z-order layout) or even 8 (if nodes are not ordered) children pointers, meaning that the Node Set representation can save 56 to 248 Bytes per 8 nodes!

Overall, our compact Sibling-Set layout results in near 8 times savings in space comparing to a well-tuned Octree implementation which stores Octree nodes individually.

Octree Optimization III: Storing Out-of-Core Octree in B+Tree. We store an out-of-core Octree in a set of B+Trees, one per each level of the Octree. For each level of the Octree, every Node Set (i.e., 8 siblings) is associated with a key which is the first sibling's Z value, and saved in a B+Tree. Each B+Tree is stored in a backend file on SSD and associated with an in-memory buffer. Each node can be addressed by searching the B+Tree of its level with the Z value of the Node Set it belongs to. Note that under this implementation, there is no child pointer for each Node Set; instead, walking down the tree is done by deriving the Z value of the child Node Set to visit, and then searching the B+Tree of the child level using the Z-value as key.

The advantages of storing Octree in B+Trees keyed by Z-value are twofold. First, it achieves fast tree construction and query performance, since B+Tree is well optimized and known for its balanced read and write performance. Second, it preserves spatial locality, since Octree nodes are stored in leaf nodes of B+Tree in Z order.

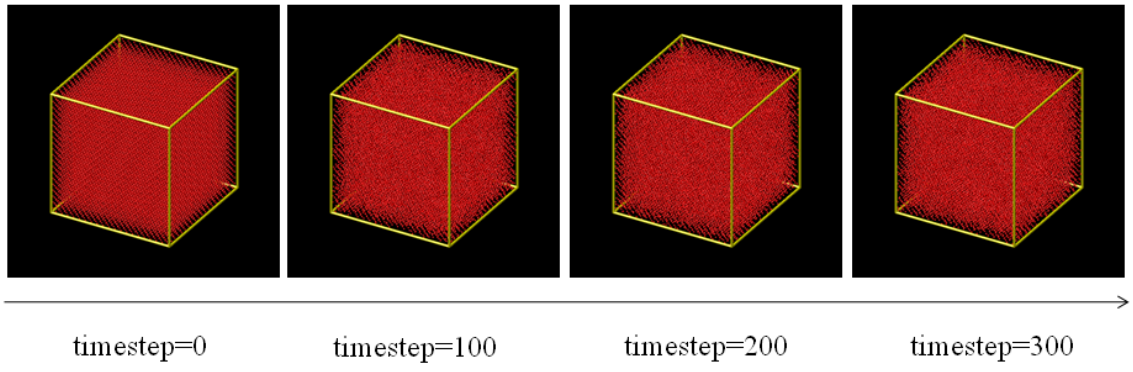


Figure 43: Visualization of sample LAMMPS atoms data.

6.5 Applications

ZStore enables online spatial index on simulation output data and can benefit many useful analytics. In this section, we describe two real-world application examples.

6.5.1 LAMMPS Application

The LAMMPS Molecular Dynamics simulation can be coupled with the Smart-Pointer [148] analytics pipeline for online data exploration. As part of the Smart-Pointer pipeline, the Bond analytics program takes as input the atoms array emitted from LAMMPS simulation, and calculates and outputs bonded atom pairs (two atoms whose distance is within a pre-defined threshold) among all atoms. A sample of LAMMPS atoms data is shown in Figure 43.

The original Bond implementation uses a two-level loop to calculate bonded atoms and has a complexity of $O(N^2)$ where N is the total number of atoms. We can leverage the RTree index provided by ZStore to accelerate Bond computation. We build a RTree index out of the atoms using the 3D coordinates of atoms as MBRs. Then for each atom, we perform a spherical query on the RTree to find those atoms whose distance is within the threshold and hence is bonded with the query atom. Both RTree construction and query is of $O(NlogN)$, so the total time complexity of Bond is reduced to $O(NlogN)$.

6.5.2 S3D Application

The S3D combustion simulation periodically outputs species data which are 22 3-dimensional double-typed arrays. Earlier we described how to use a parallel volume rendering code to visualize the species. Here we investigate another commonly used visualization on the S3D data: iso-surface extraction. An iso-surface is constructed by first finding all voxels in the volume data whose values contain a given iso-value, and then applying the classic Marching Cube algorithm [91] to generate polygons for the iso-surfaces.

The iso-surface extraction can be accelerated via the Octree index. We build an online Octree from the S3D volume data. During the Octree construction, we use a pre-defined iso-value range to guide the refinement of Octree, so that the regions which contain iso-values are indexed by Octree nodes at finer granularity. If there are multiple analytics processes, the S3D volume data are distributed by evenly distributing the number of leaf Octree nodes, so that the marching cube computation is balanced among the analytics processes. Each node in Octree contains a minimum and maximum value of its covered volume. To perform marching cube computation, the Octree is traversed from top down in breadth-first order, and the iso-value is used to filter out branches whose ranges do not to overlap with the iso-surface.

6.6 Performance Evaluation

6.6.1 Experimental Environment

We conduct experiments on the Sith cluster at Oak Ridge National Laboratory. Sith is an Opteron-based InfiniBand cluster running Linux. The system contains 40 compute nodes. Each compute node contains four 2.3 GHz 8 core AMD Opteron processors, and 64 GB of memory. Each Sith compute node contains 1.4TB of SSD partition. A software RAID, RAID-0, is used and consists of three disks where each disk is Samsung SSD 840 PRO Series. Ext4 file system is used for the SSD partition. The

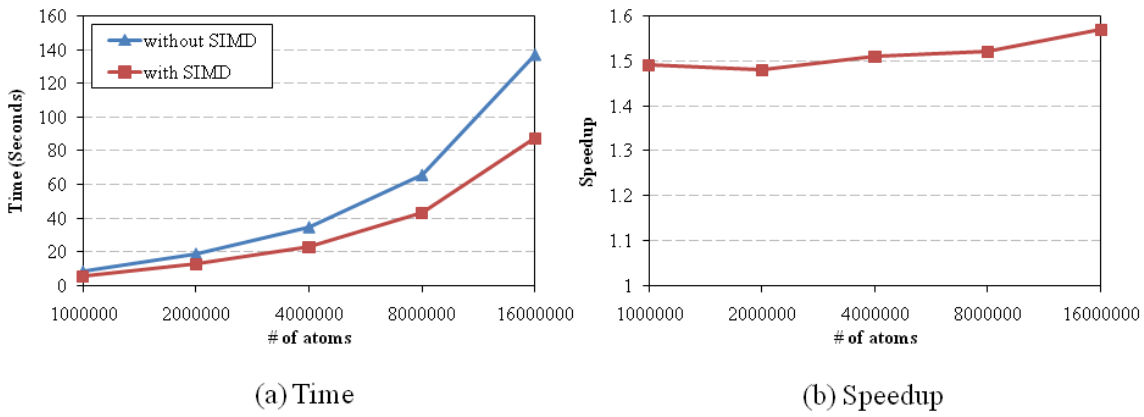
system is also configured with a 86 TB Lustre file system for scratch space.

6.6.2 LAMMPS Application

[**LAMMPS Input Setup**] We use the EAM benchmark setup from the LAMMPS benchmark suite included in the LAMMPS package. The EAM metallic solid benchmark simulates Cu metallic solid with embedded atom method (EAM) potential. The output of LAMMPS simulation is an array of atoms, in which each atom has five attributes including 3D coordinates, velocity, and ID. Figure 43 shows a sample visualization of the atoms array generated by LAMMPS EAM benchmark. We can see that the atoms are distributed in the 3D domain in a dense manner. This is in fact a representative case for LAMMPS solid and liquid simulations.

[**In-Core Bond Performance**] For the In-Core version of Bond, the RTree index is kept in DRAM. We compare two versions of the RTree implementation. The Baseline version of RTree uses the conventional node layout (that is, an array of entries), and use depth-first search during tree traversal. The SIMD version of RTree uses a column-oriented node layout (arrays of the same attributes, as described in Section 6.4), and the insertion and query routines are vectorized using SIMD instructions. Since the SIMD version evaluates multiple entries at once during query, it uses a combination of depth-first and breath-first searches (breadth first search within a node and depth first search across nodes).

We vary the number of atoms contained in the atom array from 1 million to 16 millions. Figure 44 shows the RTree construction and query time with the Baseline vs. SIMD versions. We can see that the SIMD version can outperform the Baseline version by up to 60% in all test cases. Note that the atoms array is of float type (32 bits), and each MMX register on the AMD CPU is 128 bits (so each SIMD instruction can process 4 floats at once). There are several factors which prevent us from achieving 400% speedup: first, the vectorized routines in tree insertion and



(a) Time (b) Speedup

Figure 44: RTree index construction and query time.

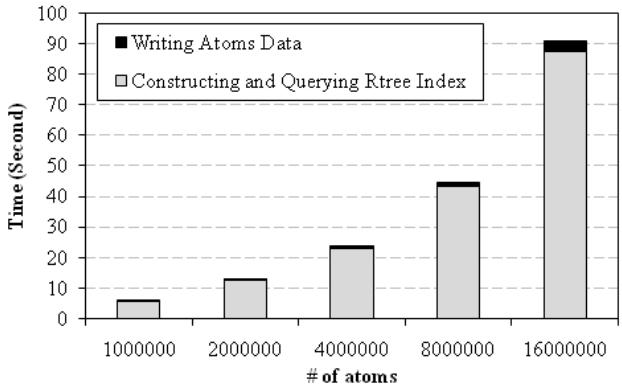


Figure 45: Bond total runtime breakdown.

query weigh about 60% of the total runtime; second, SIMD instruction is not used for processing the first few elements of an array (due to memory alignment); third, the load and store instructions (via the _mm_load_ps and _mm_store_ps intrisics) need to be used to move data between memory and MMX registers. Nevertheless, Figure 44 clearly shows the benefits of the column-oriented node layout and SIMD vectorization.

Figure 45 shows that the index construction and query time dominates the total runtime of Bond, and that the time spent in writing the atoms array to file system is less than 3% of total runtime. This is expected since the atoms array is relatively small (20MB to 320MB) under the In-Core setups.

[**Out-Of-Core Bond Performance**] For the Out-Of-Core version of Bond, the

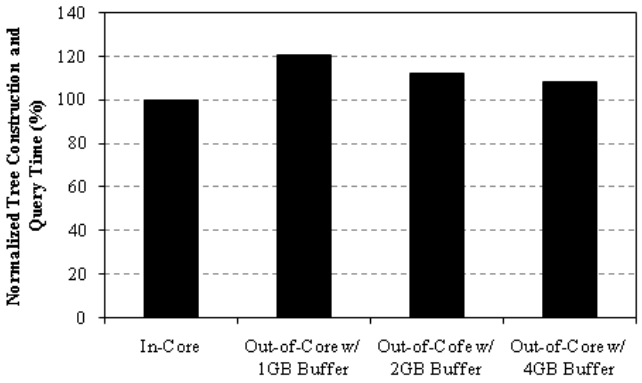


Figure 46: Performance of out-of-core Bond.

RTree index is stored in a backend file on SSD, and associated with a DRAM buffer managed under the second-chance Clock replacement policy. We are interested in the trade-off between Bond performance vs. DRAM usage. For this purpose, we run Bond to process an atom array with 1 billion atoms (20GB in total), vary the DRAM buffer size, and measure the Bond runtime. In all test cases, we fix the buffer block size to be 4KB. In order to reduce the impact of OS-level caching, we run a separate program to mmap() and hold large amounts of DRAM during the Bond run, and the buffer replacement use direct I/O to bypass buffer cache.

Figure 46 shows the Bond performance under different DRAM buffer sizes. For the 20GB atoms data, using 4GB DRAM (plus a SSD backend) can achieve performance (8% slowdown) comparable with keeping all atoms in DRAM. This means significant (5 times) savings in DRAM usage. Although the experiments shown here does not actually process data larger than the node's physical DRAM, the results still demonstrate that ZStore can enable high performance and memory efficient spatial index on the SSD-equipped deep memory hierarchy. We expect that such ability of supporting out-of-core spatial index will be increasingly pertinent to data-intensive analytics on future Exascale machines.

[**Parallel Bond Performance**] Bond is parallelized by replicating atoms in the

Table 5: Parallel Bond performance.

LAMMPS vs. Bond processes	**256 - 4**	**512 - 8**
Move Atoms from Simulation to Bond	15.9 seconds	17.3 seconds
Local RTree Construction and Query	389.5 seconds	391.9 seconds
Local Ghost Zone Calculation	6.0 seconds	5.9 seconds
Replicate Atoms in Ghost Zones	2.7 seconds	3.2 seconds
Query Atoms in Ghost Zones	5.1 seconds	5.2 seconds

ghost zones. In order to understand the end-to-end performance of LAMMPS/Bond pipeline, we run parallel LAMMPS simulation and couple it with a parallel Bond program, and use the FlexIO to couple the two. ZStore is used to build the RTree index on the fly and made the index available for Bond. We apply weak scaling: LAMMPS simulation runs on 256 and 512 processes, and the parallel Bond runs on 4 and 8 processes correspondingly (that is, the ratio of simulation to analytics processes is fixed to 64:1); Each LAMMPS simulation process emits a sub-array of atoms containing 1 million atoms (20MB in size).

Table 5 shows the time breakdown of the end-to-end latency (that is, the time from when LAMMPS simulation writes out atoms array till parallel Bond finishes processing the atoms). We can make the following observations. First, the end-to-end latency is about 423 seconds and stays almost constant when scaling up. It should be noted that the LAMMPS simulation processes uses asynchronous writes and only experience visible write latency of less than 0.1 second. The latency shown in Table 5 is hidden from the simulation as long as it outputs data less often than once every 423 seconds. Second, the majority of the latency is contributed by constructing and querying the RTree (up to 391.9 seconds). The cost of parallelizing Bond (that is, generating and replicating ghost zones), on the other hand, weighs only a small portion of the total runtime. This implies that the performance of parallel Bond is dominated by local computation.

6.6.3 S3D Application

As mentioned in Section 6.5, S3D simulation can be coupled with the iso-surface visualization to identify regions of interest. In our experiments, we run the iso-surface visualization in a controlled benchmark scenario: A 4GB S3D O2 species array (which is a sub-array of output data generated from a production run of S3D simulation) is pre-loaded into DRAM; The iso-surface visualization reads data chunk by chunk (each chunk is a 3D sub-array and 2MB in size to emulate the output of each S3D simulation process), builds a sub-Octree from each chunk, and merges into an out-of-core Octree. The out-of-core Octree is saved on a backend file on SSD and associated with 1GB DRAM buffer with 4KB block size. Although we do not run the iso-surface visualization to process live S3D simulation output data, the benchmark results shown here still can demonstrate the runtime behavior of single-process iso-surface visualization and yield useful insights.

We compare four different versions of the iso-surface visualization, all of which uses the Marching Cube algorithm as mentioned earlier:

1) Original Marching Cube: scan the volume data for the queried iso-value;

2) Marching Cube with a full Octree of height 3;

3) Marching Cube with a full Octree of height 4

4) Marching Cube with an Octree trimmed according to the querying iso-value.

For cases 2 and 3, a full Octree is constructed. In the forth case, the Octree is constructed by trimming the octants which do not contain the given iso-value. This represents a case where prior knowledge of the iso-value(s) to be queried is used to guide the construction of the Octree index.

Figure 47 shows the runtime of Octree construction and Marching Cube algorithm to calculate one iso-surface. For the case of the original Marching Cube, there is no tree construction cost and generating one iso-surface takes about 209 seconds. When using a full Octree of height 3, it takes 160 seconds to construct the Octree index

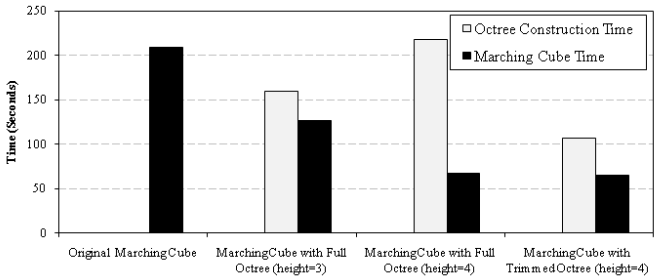


Figure 47: Performance of Iso-Surface using Marching Cube algorithm.

and 127 seconds to generate the iso-surface using the Octree. Constructing a full Octree of height 4 takes more time (218 seconds), but the finer grained Octree index helps reduce the iso-surface runtime to 67 seconds. Trimmed Octree is built based on prior knowledge of iso-value to be queried. We can see that incorporating such prior knowledge not only reduces Octree construction time compared to the full Octree, but also achieve the same query time with the full Octree of the same height.

The results show that using Octree can improve query performance significantly. The additional cost of tree construction can be well amortized as long as a sufficient number of queries are performed. We expect that potential use cases for Octree-based iso-surface visualization are interactive queries and collaborative data exploration.

6.7 Related Work

Distributed Spatial Index. Spatial indices are widely used to accelerate query and visualization on massive scientific data sets. Early work such as Master Client RTree [125] uses a single central index server for distributed indexing and query. To further improves scalability, [100] proposes a two-level hierarchical indexing architecture for distributed data sets, in which a central master server holds the top portion of the RTree index and routes queries to other servers each of which holds a portion of the data set and corresponding local index. [101] addresses scalability by organizing the index servers via a Distributed Hash Table. Similarly, [32] uses DHT to

organize distributed RTree indices but targets the Cloud environment. In ZStore, we use partial replication for distributed index and query. This is easy to implement and has been shown to be generally applicable to different spatial indices; and the cost of maintaining consistency among replicas can be avoided for scientific data which is read-only. In the future, we plan to investigate other index organization schemes.

Tree-Based Parallel Computing. Spatial trees are also extensively used in parallel computing for N-Body simulation, Adaptive Mesh Refinement, and Machine Learning. [75] presents a general framework for constructing parallel trees and using the trees for a variety of kernel summation algorithms. [74] proposes a massively parallel FMM algorithm using an adaptive parallel Octree. [16] proposes algorithms for parallel KD tree construction. Our work focuses on design a general framework for constructing and querying spatial index from live simulation data in an online streaming fashion. The parallelization techniques proposed in previous work can be leveraged to further improve index performance.

In Situ Indexing and Query. Work on in situ bitmap indexing [69] demonstrate the feasibility of generating bitmap indices online and in parallel. Since the bitmap index construction process is data parallel, ZStore can easily support it using the in-transit index construction workflow. [72] jointly applies compression and in-network aggregation to construct index from simulation output data. ZStore differs in its flexibility in controlling data distribution and generality to a variety of spatial indices.

SSD for HPC. Previous work has explored the use of SSD as part of HPC storage system, mainly as a cache for HDD [31, 149]. The Gordon supercomputer [58] at SDSC is one of the real machine installation and shows advantages for data-intensive workloads [57]. Others [83, 119] propose to install SSD onto a set of staging nodes to help buffer bursty I/O traffic. [65] uses simulation to demonstrate the benefits of node-local SSDs for out-of-core scientific computing. [140] proposes to aggregate distributed SSDs as a memory partition exposed via a malloc()-like interface. ZStore

rides this architectural trend and provides a buffer manager utility to allow out-of-core multi-dimensional arrays and indices on SSD-equipped deep memory hierarchy.

6.8 Conclusions

6.8.1 Summary

In this chapter, we introduce ZStore which provides a general and scalable framework to construct spatial indices from live simulation output data and offer the index to analytics for answering various spatial queries. ZStore uses a flexible in-transit index construction workflow embedded in the I/O path which leverages distributed resources to build index in a streaming manner. The workflow is highly customizable to allow application-specific control in data distribution. ZStore also provides a SSD-optimized buffer management utility for building out-of-core index on deep memory hierarchy. We implement two representative spatial indices: RTree and Octree with ZStore, and for each index we propose novel optimizations which significantly improve performance and memory efficiency. We demonstrate the utility of ZStore with two leadership scientific applications: LAMMPS and S3D.

6.8.2 Lessons Learned

The use of online spatial index presents an interesting way of improving the end-to-end performance of coupled simulation and analytics: simulation output data is "prepared" according to the intention and interest of analytics, so that the prepared data can be processed by analytics much faster than the raw data. There is apparently a trade-off between the additional cost of data preparation (index construction, storage and dissemination) vs. the improvement in analytics (query acceleration). Currently, ZStore does not estimate such trade-off. The applications shown above are all cases where the cost pays off. It would be an interesting topic to explore the trade-off between cost vs. performance gains in building online indices, and make adaptive decisions about whether or which index should be used.

Another lesson we have learned from ZStore is that flexibility is a good way to accommodate the differences among various use cases. For example, the RTree and Octree implementation require different index construction and distribution strategies. By using a callback-based framework, ZStore provides sufficient flexibility in implementing those different use cases. And we expect that the same framework can be applied to other indices such as bitmap index.

CHAPTER VII

CONCLUSION

7.1 Conclusions

Fast analytics on Big Data is essential to drive today's scientific discoveries. The middleware solutions developed as part of this dissertation make it possible for domain scientists to perform a wide range of analytics on live simulation output data, ranging from simulation monitoring, data diagnostics, reorganization of data layout for improved performance in storage or post-processing action, meta-data annotation, to data visualization. The capability of performing online analytics in a timely and cost-efficient manner enhances the process of scientific discovery and helps the Scientific Computing community in coping with the massive volumes of scientific data produced by current and future High End Computing machines.

The dissertation makes the following concrete technical contributions.

1) The PreDatA middleware for Preparatory Data Analytics on large scale simulation output data, offers a MapReduce-like model for programming application-specific operations on streaming data. With this model, users can exploit the distributed computational power along the I/O path to perform online analytics on high end machines and before data is placed into storage. Performance evaluations with real-world peta-scale applications on up to 16384 cores demonstrate that PreDatA is useful for data pre-processing, runtime data analysis and inspection, as well as for data exchange between concurrently running simulations. Using the PreDatA solution can improve the execution times of large-scale simulations, provide timely insight into their output data, and improve the read performance seen by data post-processing steps for the output files being generated.

2) Experimental results and performance modeling reveals that the placement of online data analytics onto the underlying resources of a high end machine can significantly impact the end-to-end performance of the I/O pipelines used by simulations. Building on the PreDatA approach, the FlexIO middleware offers additional functionality to improve the degree of flexibility seen for analytics placement. In particular, it provides high performance, memory efficient intra- and inter-node data transports; it supports complex data exchange patterns; and it presents to developers high level interfaces for specifying I/O pipelines and their component interactions. With these interfaces and underlying automation for choosing appropriate transports and data exchanges, it makes changes in analytics placement transparent to simulation and analytics codes. Placement policies built on top of FlexIO can exploit its location flexibility to tune I/O pipeline performance and overheads like data movement cost. Experiments show that leveraging the flexibility enabled by FlexIO to tune placement can improve total execution time by up to 30% compared to alternative solutions, with benefits more evident at larger scales.

3) The Goldrush runtime resource management methods leverage idle resources on the compute nodes used by a simulation to run online data analytics 'close' to the data being generated, thus reducing data movements and their associated costs. GoldRush does so by harvesting otherwise-wasted, idle resources on compute nodes using fine-grained, predictive, on-node scheduling in ways that incurs negligible runtime overheads and minimizes interference between the simulation and analytics. Such scheduling involves detecting sufficiently large periods of resource idleness and identifying and then avoiding the potential causes of on-node resource contention. Experiments with representative applications at large scales (up to 12288 cores on Hopper Cray XE6) show that resources harvested by GoldRush can be used to perform useful analytics, significantly improving resource efficiency, reducing data movement costs, and posing negligible impact on simulations.

4) The "ZStore" framework supports the online construction of spatial indices from live simulation output data, the goal being to make these indices available to analytics for answering spatial queries of interest. ZStore offers an in-transit index construction workflow embedded in the I/O path to build an index in a streaming manner. The workflow is customizable for application-specific control over data distribution. It provides a SSD-optimized buffer management utility for building out-of-core index structures on emerging SSD-equipped machine nodes, to support their deeper memory hierarchies. Two representative spatial indices are implemented with ZStore: a RTree and an Octree, and for each index, novel optimizations are applied to improve performance and memory efficiency. ZStore is used to support queries of interest for two large scale scientific applications and their analytics workflows, demonstrating its utility and importance, with initial performance results showing the viability of in-memory and out-of-core index construction for large scale scientific data.

Overall, the work presented in this dissertation confirms the following thesis statement:

I/O middleware offering methods for efficient data movement, flexible analytics placement, interference management, and support for online spatial indices can enable high performance and resource-efficient online data analytics to process massive simulation output data at large scale.

7.2 Future Work

7.2.1 Online Data Analytics on Heterogeneous Platforms.

One interesting extension of our work is to support online scientific data analytics in heterogeneous environment, inclding for machines with attached accelerators like the General-Purpose GPUs (GPGPU). This is because GPGPUs have become increasingly pervasive in today's HPC platforms, including large installations on some

of today's top-ranked supercomputers like the Titan Cray XK7 and China's Tianhe-2. GPGPUs provide enormous computational power that can significantly accelerate both scientific simulation and analytics, and therefore, present a valuable opportunity to further improve the end-to-end performance of the online analytics pipelines desired for high end codes. To run online scientific data analytics on heterogeneous platforms, however, analytics must be carefully mapped and scheduled along with the simulation, so that i) computation loads are balanced among heterogeneous resources; ii) overall resource utilization is high; and iii) data movement costs are minimized and/or overlapped with computation. Part of our ongoing work is to extend the GoldRush runtime for scheduling simulation and analytics to synergistically share GPUs.

Another source of heterogeneity comes from ongoing changes in memory technology. Non-Volatile Memories such as Phase Changing Memory are a promising solution to the decreasing per-core amounts of DRAM present in high end servers. Their adoption will lead to increased levels of depth and heterogeneity in servers' memory hierarchies, in terms of their performance, power, and reliability characteristics. This calls for revisiting the design and implementation of online data analytics to fully exploit heterogeneous memory resources. The SSD-aware spatial index supported by ZStore offers one way forward for carrying out such work.

7.2.2 Combining Online and Offline Analytics

This thesis has shown online data analytics to be useful to real-world science applications. A future step is to combine online and offline analytics to create even more useful tools for scientific discovery. On the one hand, online analytics delivers fast insights from data, but on the other hand, it is constrained by limited resources and the consequently small time windows over which it is performed. Offline analytics, however, can operate over long time scales to deeply process data, and it can process data multiple times and in forward or backward order. The ability to combine the

merits of both types of analytics may further improve the overall scientific discovery process. One example is to use online analytics to prepare data (e.g., via layout re-organization, indexing, or early data reduction) so that the later offline analytics can be accelerated. Another example is to let the offline analytics provide feedback information derived from historical data to steer how online analytics are performed (e.g., to refine the regions of interest to be tracked online).

REFERENCES

[1] *Combining in-situ and in-transit processing to enable extreme-scale scientific analysis*, 2012.

[2] "Gromacs." http://www.gromacs.org/, August 2012.

[3] *In-situ Feature-based Objects Tracking for Large-Scale Scientific Simulations*, 2012.

[4] "Papi: Performance application programming interface." http://icl.cs.utk.edu/papi/, September 2012.

[5] "Scotch library." http://www.labri.fr/perso/pelegrin/scotch/, September 2012.

[6] "Xpmem." http://code.google.com/p/xpmem/, September 2012.

[7] "Hopper cray xe6 at nersc." http://www.nersc.gov/systems/hopper-cray-xe6/, August 2013.

[8] "Nas parallel benchmarks." http://www.nas.nasa.gov/publications/npb.html, August 2013.

[9] "Vampir performance tool." http://www.vampir.eu/, August 2013.

[10] Abbasi, H., Eisenhauer, G., Wolf, M., Schwan, K., and Klasky, S., "Just in time: adding value to the io pipelines of high performance applications with jitstaging," in *HPDC*, pp. 27–36, 2011.

[11] Abbasi, H., Lofstead, J., Zheng, F., Klasky, S., Schwan, K., and Wolf, M., "Extending i/o through high performance data services," in *Cluster Computing*, (Luoisiana, LA), IEEE International, September 2009.

[12] Abbasi, H., Wolf, M., Eisenhauer, G., Klasky, S., Schwan, K., and Zheng, F., "Datastager: scalable data staging services for petascale applications," in *HPDC*, pp. 39–48, 2009.

[13] Abbasi, H., Wolf, M., Schwan, K., Eisenhauer, G., and Hilton, A., "Xchange: coupling parallel applications in a dynamic environment," in *CLUSTER*, pp. 471–480, 2004.

[14] ADIOS, "Adios: Adaptive i/o system." http://www.olcf.ornl.gov/center-projects/adios/, September 2012.

[15] Ailamaki, A., Kantere, V., and Dash, D., "Managing scientific data," *Commun. ACM*, vol. 53, no. 6, pp. 68–78, 2010.

[16] Al-Furaih, I., Aluru, S., Goil, S., and Ranka, S., "Parallel construction of multidimensional binary search trees," in *Proceedings of the 10th International Conference on Supercomputing*, ICS '96, (New York, NY, USA), pp. 205–212, ACM, 1996.

[17] Ali, N., Carns, P. H., Iskra, K., Kimpe, D., Lang, S., Latham, R., Ross, R. B., Ward, L., and Sadayappan, P., "Scalable i/o forwarding framework for high-performance computing systems," in *CLUSTER*, pp. 1–10, 2009.

[18] Amiri, K., Petrou, D., Ganger, G. R., and Gibson, G. A., "Dynamic function placement for data-intensive cluster computing," in *ATEC '00: Proceedings of the annual conference on USENIX Annual Technical Conference*, 2000.

[19] Andrade, H., Gedik, B., Wu, K.-L., and Yu, P. S., "Processing high data rate streams in system s," *J. Parallel Distrib. Comput.*, vol. 71, no. 2, pp. 145–156, 2011.

[20] B. Whitlock, J. Favre, J. M., "Parallel in situ coupling of a simulation with a fully featured visualization system," 2011.

[21] Bernat, A. R. and Miller, B. P., "Anywhere, any-time binary instrumentation," in *Proceedings of the 10th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools*, PASTE '11, (New York, NY, USA), pp. 9–16, ACM, 2011.

[22] Beynon, M. D., Ferreira, R., Kurç, T. M., Sussman, A., and Saltz, J. H., "Datacutter: Middleware for filtering very large scientific datasets on archival storage systems," in *MSST*, pp. 119–134, 2000.

[23] Biddiscombe, J., Soumagne, J., Oger, G., Guibert, D., and Piccinali, J.-G., "Parallel computational steering and analysis for hpc applications using a paraview interface and the hdf5 dsm virtual file driver," in *EGPGV*, pp. 91–100, 2011.

[24] BOINC.

[25] Brightwell, R., Hudson, T., Pedretti, K. T., Riesen, R., and Underwood, K. D., "Implementation and performance of portals 3.3 on the cray xt3," in *CLUSTER*, pp. 1–10, 2005.

[26] Buck, J. B., Watkins, N., LeFevre, J., Ioannidou, K., Maltzahn, C., Polyzotis, N., and Brandt, S., "Scihadoop: Array-based query processing in hadoop," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, (New York, NY, USA), pp. 66:1–66:11, ACM, 2011.

[27] Buntinas, D., Goglin, B., Goodell, D., Mercier, G., and Moreaud, S., "Cache-efficient, intranode, large-message mpi communication with mpich2-nemesis," in *Proceedings of the 2009 International Conference on Parallel Processing*, ICPP '09, (Washington, DC, USA), pp. 462–469, IEEE Computer Society, 2009.

[28] Carns, P., Lang, S., Ross, R., Vilayannur, M., Kunkel, J., and Ludwig, T., "Small-file access in parallel file systems," in *IPDPS*, 2009.

[29] Center, S., "Scidac scientific data management center." https://sdm.lbl.gov/sdmcenter/, September 2009.

[30] Chacón, L., "A non-staggered, conservative, $\nabla \dot{s} B \rightarrow = 0$, finite-volume scheme for 3D implicit extended magnetohydrodynamics in curvilinear geometries," *Computer Physics Communications*, vol. 163, pp. 143–171, Nov. 2004.

[31] Chen, F., Koufaty, D. A., and Zhang, X., "Hystor: Making the best use of solid state drives in high performance storage systems," in *Proceedings of the International Conference on Supercomputing*, ICS '11, (New York, NY, USA), pp. 22–32, ACM, 2011.

[32] Chen, G., Vo, H. T., Wu, S., Ooi, B. C., and Özsu, M. T., "A framework for supporting dbms-like indexes in the cloud," *PVLDB*, vol. 4, no. 11, pp. 702–713, 2011.

[33] Childs, H., "Architectural challenges and solutions for petascale postprocessing," *J. Phys.: Conf. Ser.*, vol. 78, 2007.

[34] Childs, H., Pugmire, D., Ahern, S., Whitlock, B., Howison, M., Prabhat, Weber, G. H., and Bethel, E. W., "Extreme scaling of production visualization software on diverse architectures," *IEEE Computer Graphics and Applications*, vol. 30, no. 3, pp. 22–31, 2010.

[35] Chou, J., Howison, M., Austin, B., Wu, K., Qiang, J., Bethel, E. W., Shoshani, A., Rübel, O., Prabhat, and Ryne, R. D., "Parallel index and query for large scale data analysis," in *SC*, p. 30, 2011.

[36] Cray, I., "Using cray performance analysis tools," June 2010.

[37] Cudre-Mauroux, P., Kimura, H., Lim, K.-T., Rogers, J., Simakov, R., Soroush, E., Velikhov, P., Wang, D. L., Balazinska, M., Becla, J., DeWitt, D., Heath, B., Maier, D., Madden, S., Patel, J., Stonebraker, M., and Zdonik, S., "A demonstration of scidb: A science-oriented dbms," *Proc. VLDB Endow.*, vol. 2, pp. 1534–1537, Aug. 2009.

[38] Dean, J. and Ghemawat, S., "Mapreduce: simplified data processing on large clusters," in *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, (Berkeley, CA, USA), pp. 10–10, USENIX Association, 2004.

[39] Deelman, E. and Chervenak, A., "Data management challenges of data-intensive scientific workflows," in *CCGRID '08: Proceedings of the 2008 Eighth IEEE International Symposium on Cluster Computing and the Grid*, (Washington, DC, USA), pp. 687–692, IEEE Computer Society, 2008.

[40] Deelman, E., Singh, G., hui Su, M., Blythe, J., Gil, A., Kesselman, C., Mehta, G., Vahi, K., Berriman, G. B., Good, J., Laity, A., Jacob, J. C., and Katz, D. S., "Pegasus: a framework for mapping complex scientific workflows onto distributed systems," *Scientific Programming Journal*, vol. 13, pp. 219–237, 2005.

[41] Docan, C., Parashar, M., Cummings, J., Podhorszki, N., and Klasky, S., "Experiments with Memory-to-Memory Coupling for End-to-End Fusion Simulation Workflows.," Tech. Rep. TR-104, Center for Autonomic Computing (CAC), Rutgers University, Piscataway, NJ, USA, July 2009.

[42] Docan, C., Parashar, M., and Klasky, S., "Dart: a substrate for high speed asynchronous data io," in *HPDC*, pp. 219–220, 2008.

[43] Dorier, M., Antoniu, G., Cappello, F., Snir, M., and Orf, L., "Damaris: How to efficiently leverage multicore parallelism to achieve scalable, jitter-free i/o," *2013 IEEE International Conference on Cluster Computing (CLUSTER)*, vol. 0, pp. 155–163, 2012.

[44] E. R. Hawkes, R. Sankaran, J. C. S. and Chen, J. H., "Direct numerical simulation of turbulent combustion: fundamental insights towards predictive models," *Journal of Physics: Conference Series*, vol. 16, pp. 65–79, 2005.

[45] Eisenhauer, G., "Evpath: event transport middleware layer." http://www.cc.gatech.edu/systems/projects/EVPath/, September 2009.

[46] Eisenhauer, G., Bustamante, F. E., and Schwan, K., "Native data representation: An efficient wire format for high-performance distributed computing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 13, no. 12, pp. 1234–1246, 2002.

[47] Eisenhauer, G., Wolf, M., Abbasi, H., Klasky, S., and Schwan, K., "A type system for high performance communication and computation," in *e-Science Workshops*, pp. 183–190, 2011.

[48] Feng, W.-C. and Scogland, T., "The Green500 List: Year One," in *5th IEEE Workshop on High-Performance, Power-Aware Computing (in conjunction with the 23rd International Parallel & Distributed Processing Symposium)*, (Rome, Italy), May 2009.

[49] for Computational Sciences, N. C., "Smoky cluster, month =."

[50] Fryxell, B., Olson, K., Ricker, P., Timmes, F. X., Zingale, M., Lamb, D. Q., MacNeice, P., Rosner, R., Truran, J. W., and Tufo,

H., "Flash: An adaptive mesh hydrodynamics code for modeling astrophysical thermonuclear flashes," *The Astrophysical Journal Supplement Series*, vol. 131, no. 1, pp. 273–334, 2000.

[51] Gerndt, A., Hentschel, B., Wolter, M., Kuhlen, T., and Bischof, C., "Viracocha: An efficient parallelization framework for large-scale cfd post-processing in virtual environments," in *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, SC '04, (Washington, DC, USA), pp. 50–, IEEE Computer Society, 2004.

[52] Giacomoni, J., Moseley, T., and Vachharajani, M., "Fastforward for efficient pipeline parallelism: a cache-optimized concurrent lock-free queue," PPoPP '08, 2008.

[53] Gray, J., Liu, D. T., Nieto-Santisteban, M., Szalay, A., DeWitt, D. J., and Heber, G., "Scientific data management in the coming decade," *SIGMOD Rec.*, vol. 34, no. 4, pp. 34–41, 2005.

[54] Gu, W., Eisenhauer, G., Schwan, K., and Vetter, J. S., "Falcon: On-line monitoring for steering parallel programs," *Concurrency - Practice and Experience*, vol. 10, no. 9, pp. 699–736, 1998.

[55] Guttman, A., "R-trees: A dynamic index structure for spatial searching," in *SIGMOD'84, Proceedings of Annual Meeting, Boston, Massachusetts, June 18-21, 1984* (Yormark, B., ed.), pp. 47–57, ACM Press, 1984.

[56] Hadjieleftheriou, M., Manolopoulos, Y., Theodoridis, Y., and Tsotras, V. J., "R-trees - a dynamic index structure for spatial searching," in *Encyclopedia of GIS*, pp. 993–1002, 2008.

[57] He, J., Bennett, J., and Snavely, A., "Dash-io: An empirical study of flash-based io for hpc," in *Proceedings of the 2010 TeraGrid Conference*, TG '10, (New York, NY, USA), pp. 10:1–10:8, ACM, 2010.

[58] He, J., Jagatheesan, A., Gupta, S., Bennett, J., and Snavely, A., "Dash: A recipe for a flash-based data intensive supercomputer," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, (Washington, DC, USA), pp. 1–11, IEEE Computer Society, 2010.

[59] Hoefler, T., Schneider, T., and Lumsdaine, A., "Characterizing the influence of system noise on large-scale applications by simulation," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, (Washington, DC, USA), pp. 1–11, IEEE Computer Society, 2010.

[60] Huston, L., Sukthankar, R., Wickremesinghe, R., Satyanarayanan, M., Ganger, G. R., Riedel, E., and Ailamaki, A., "Diamond: A storage

architecture for early discard in interactive search," in *Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, 2004.

[61] Jay Lofstead, Ron Oldfield, T. K. C. R., "Extending scalability of collective io through nessie and stagin," in *PDSW*, 2011.

[62] Jay Lofstead, Ron Oldfield, T. K. C. R., "Developing integrated data services for cray systems with a gemini interconnect," in *Cray User Group Meeting*, 2012.

[63] Jin, T., Zhang, F., Sun, Q., Bui, H., Parashar, M., Yu, H., Klasky, S., Podhorszki, N., and Abbasi, H., "Using cross-layer adaptations for dynamic data management in large scale coupled scientific workflows," in *SC*, p. 74, 2013.

[64] Jones, C., Ma, K.-L., Sanderson, A., and Jr, L. R. M., "Visual interrogation of gyrokinetic particle simulations," *J. Phys.: Conf. Ser.*, vol. 78, no. 012033, p. 6, 2007.

[65] Jung, M., Wilson, III, E. H., Choi, W., Shalf, J., Aktulga, H. M., Yang, C., Saule, E., Catalyurek, U. V., and Kandemir, M., "Exploring the future of out-of-core computing with compute-local non-volatile memory," in *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '13, (New York, NY, USA), pp. 75:1–75:11, ACM, 2013.

[66] keng Liao, W. and Choudhary, A. N., "Dynamically adapting file domain partitioning methods for collective i/o based on underlying parallel file system locking protocols," in *SC*, p. 3, 2008.

[67] Kenneth Moreland, Ron Oldfield, P. M. S. J. N. P. V. V. N. F. C. D. M. P. M. H. M. E. P. and Klasky, S., "Examples of in transit visualization," in *Petascale Data Analytics: Challenges and Opportunities (PDAC-11), In Cooperation with ACM/IEEE SC11*, 2011.

[68] Khandekar, R., Hildrum, K., Parekh, S., Rajan, D., Wolf, J., Wu, K.-L., Andrade, H., and Gedik, B., "Cola: optimizing stream processing applications via graph partitioning," in *Proceedings of the 10th ACM/IFIP/USENIX International Conference on Middleware*, 2009.

[69] Kim, J., Abbasi, H., Chacón, L., Docan, C., Klasky, S., Liu, Q., Podhorszki, N., Shoshani, A., and Wu, K., "Parallel in situ indexing for data-intensive computing," in *LDAV*, pp. 65–72, 2011.

[70] Klasky, S., Ethier, S., Lin, Z., Martins, K., McCune, D., and Samtaney, R., "Grid -based parallel data streaming implemented for the gyrokinetic toroidal code," in *SC '03*, 2003.

[71] Laboratory, S. N., "Cth shock physics." http://www.sandia.gov/CTH/, January 2012.

[72] Lakshminarasimhan, S., Boyuka, D. A., Pendse, S. V., Zou, X., Jenkins, J., Vishwanath, V., Papka, M. E., and Samatova, N. F., "Scalable in situ scientific data encoding for analytical query processing," in *Proceedings of the 22Nd International Symposium on High-performance Parallel and Distributed Computing*, HPDC '13, (New York, NY, USA), pp. 1–12, ACM, 2013.

[73] Lakshminarasimhan, S., Shah, N., Ethier, S., Klasky, S., Latham, R., Ross, R., and Samatova, N. F., "Compressing the incompressible with isabela: In-situ reduction of spatio-temporal data," in *Euro-Par (1)*, pp. 366–379, 2011.

[74] Lashuk, I., Chandramowlishwaran, A., Langston, H., Nguyen, T.-A., Sampath, R., Shringarpure, A., Vuduc, R., Ying, L., Zorin, D., and Biros, G., "A massively parallel adaptive fast multipole method on heterogeneous architectures," in *Proc. ACM/IEEE Conf. Supercomputing (SC)*, (Portland, OR, USA), November 2009.

[75] Lee, D., Vuduc, R., and Gray, A. G., "A distributed kernel summation framework for general-dimension machine learning," in *Proc. SIAM Int'l. Conf. Data Mining (SDM)*, (Anaheim, CA, USA), April 2012.

[76] Lee, J.-Y. and Sussman, A., "High performance communication between parallel programs," in *IPDPS*, 2005.

[77] Lee, J., Ross, R. B., Atchley, S., Beck, M., and Thakur, R., "Mpi-io/l: efficient remote i/o for mpi-io via logistical networking," in *IPDPS*, 2006.

[78] Lee, J., Ross, R. B., Thakur, R., Ma, X., and Winslett, M., "Rfs: efficient and flexible remote file access for mpi-io," in *CLUSTER*, pp. 71–81, 2004.

[79] Lee, M. and Schwan, K., "Region scheduling: Efficiently using the cache architectures via page-level affinity," in *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, (New York, NY, USA), pp. 451–462, ACM, 2012.

[80] Li, D., de Supinski, B. R., Schulz, M., Cameron, K. W., and Nikolopoulos, D. S., "Hybrid mpi/openmp power-aware computing," in *IPDPS*, pp. 1–12, 2010.

[81] Li, M., Vazhkudai, S. S., Butt, A. R., Meng, F., Ma, X., Kim, Y., Engelmann, C., and Shipman, G., "Functional partitioning to optimize end-to-end performance on many-core architectures," SC '10, (Washington, DC, USA), pp. 1–12, IEEE Computer Society, 2010.

[82] Litzkow, M. J., Livny, M., and Mutka., M. W., "Condor-a hunter of idle workstations," ICDCS, 1988.

[83] Liu, N., Cope, J., Carns, P. H., Carothers, C. D., Ross, R. B., Grider, G., Crume, A., and Maltzahn, C., "On the role of burst buffers in leadership-class storage systems," in *MSST*, pp. 1–11, 2012.

[84] Liu, Y., Vijayakumar, N., and Plale, B., "Stream processing in data-driven computational science," in *GRID '06: Proceedings of the 7th IEEE/ACM International Conference on Grid Computing*, (Washington, DC, USA), pp. 160–167, IEEE Computer Society, 2006.

[85] Lofstead, J., Klasky, S., Schwan, K., Podhorszki, N., and Jin, C., "Flexible io and integration for scientific codes through the adaptable io system (adios)," in *CLADE 2008 at HPDC*, (Boston, Massachusetts), ACM, June 2008.

[86] Lofstead, J., Liu, Q., Klasky, S., Booth, M., Oldfield, R., Schwan, K., and Wolf, M., "High performance io on busy systems," in *In Proceedings of PDSW 2009 at Supercomputing 2009, submitted*, 2009.

[87] Lofstead, J., Zheng, F., Klasky, S., and Schwan, K., "Input/output apis and data organization for high performance scientific computing," in *In Proceedings of PDSW 2008 at Supercomputing 2008*, 2008.

[88] Lofstead, J., Zheng, F., Klasky, S., and Schwan, K., "Adaptable, metadata rich io methods for portable high performance io," in *In IPDPS'09, May 25-29, Rome, Italy*, 2009.

[89] Lofstead, J. F., Polte, M., Gibson, G. A., Klasky, S., Schwan, K., Oldfield, R., Wolf, M., and Liu, Q., "Six degrees of scientific data: reading patterns for extreme scale science io," in *HPDC*, pp. 49–60, 2011.

[90] Lofstead, J. F., Zheng, F., Liu, Q., Klasky, S., Oldfield, R., Kordenbrock, T., Schwan, K., and Wolf, M., "Managing variability in the io performance of petascale storage systems," in *SC*, pp. 1–12, 2010.

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

[92] Lu, Q., Lin, J., Ding, X., Zhang, Z., Zhang, X., and Sadayappan, P., "Soft-olp: Improving hardware cache performance through software-controlled object-level partitioning," in *Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques*, PACT '09, (Washington, DC, USA), pp. 246–257, IEEE Computer Society, 2009.

[93] Ludäscher, B., Altintas, I., Berkley, C., Higgins, D., Jaeger, E., Jones, M., Lee, E. A., Tao, J., and Zhao, Y., "Scientific workflow management and the kepler system," *Concurrency and Computation: Practice and Experience*, vol. 18, no. 10, pp. 1039–1065, 2006.

[94] M. Hereld, M. E. Papka, V. V., "Toward simulation-time data analysis and i/o acceleration on leadership-class systems," in *LDAV*, 2011.

[95] Ma, X., Lee, J., and Winslett, M., "High-level buffering for hiding periodic output cost in scientific simulations," *IEEE Trans. Parallel Distrib. Syst.*, vol. 17, no. 3, pp. 193–204, 2006.

[96] Mars, J., Vachharajani, N., Hundt, R., and Soffa, M. L., "Contention aware execution: Online contention detection and response," in *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '10, (New York, NY, USA), pp. 257–265, ACM, 2010.

[97] Mehta, D. P. and Sahni, S., *Handbook of Algorithms and Data Structures*. Chapman and Hall/CRC, 2004.

[98] Mohr, B., Malony, A. D., Shende, S., and Wolf, F., "Design and prototype of a performance tool interface for openmp," *J. Supercomput.*, vol. 23, pp. 105–128, Aug. 2002.

[99] Moretti, C., Bulosan, J., Thain, D., and Flynn, P. J., "All-pairs: An abstraction for data-intensive cloud computing," in *IPDPS*, pp. 1–11, 2008.

[100] Nam, B. and Sussman, A., "Spatial indexing of distributed multidimensional datasets," in *CCGRID*, pp. 743–750, 2005.

[101] Nam, B. and Sussman, A., "Dist: fully decentralized indexing for querying distributed multidimensional datasets," in *IPDPS*, 2006.

[102] Nathan Fabian, Kenneth Moreland, D. T. A. B. P. M. B. G. M. R. K. J., "The paraview coprocessing library: A scalable, general purpose in situ visualization library," in *LDAV*, 2011.

[103] Nisar, A., keng Liao, W., and Choudhary, A. N., "Scaling parallel i/o performance through i/o delegate and caching system," in *SC*, p. 9, 2008.

[104] Oldfield, R. and Kotz, D., "Armada: A parallel i/o framework for computational grids," *Future Gener. Comput. Syst.*, vol. 18, pp. 501–523, Mar. 2002.

[105] Oldfield, R. and Kotz, D., "Improving data access for computational grid applications," *Cluster Computing*, vol. 9, no. 1, pp. 79–99, 2006.

[106] Oldfield, R., Sjaardema, G. D., II, G. F. L., and Kordenbrock, T., "Trilinos i/o support (trios)," *Scientific Programming*, vol. 20, no. 2, pp. 181–196, 2012.

[107] Oldfield, R., Ward, L., Riesen, R., Maccabe, A. B., Widener, P., and Kordenbrock, T., "Lightweight i/o for scientific applications," in *CLUSTER*, 2006.

[108] Oldfield, R. A., Arunagiri, S., Teller, P. J., Seelam, S., Varela, M. R., Riesen, R., and Roth, P. C., "Modeling the impact of checkpoints on next-generation systems," *Mass Storage Systems and Technologies, IEEE / NASA Goddard Conference on*, vol. 0, pp. 30–46, 2007.

[109] Park, S. and Shen, K., "A performance evaluation of scientific i/o workloads on flash-based ssds," in *In Workshop on Interfaces and Architectures for Scientific Data Storage at Cluster 2009*, 2009.

[110] PDSI, "Scidac petascale data storage institute." http://www.pdsi-scidac.org/, September 2009.

[111] Peterka, T., Ross, R., Nouanesengsy, B., Lee, T.-Y., Shen, H.-W., Kendall, W., and Huang, J., "A study of parallel particle tracing for steady-state and time-varying flow fields," in *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium*, IPDPS '11, (Washington, DC, USA), pp. 580–591, IEEE Computer Society, 2011.

[112] Piernas, J., Nieplocha, J., and Felix, E. J., "Evaluation of active storage strategies for the lustre parallel file system," in *SC*, (New York, NY, USA), pp. 1–10, ACM, 2007.

[113] Pietzuch, P., Ledlie, J., Shneidman, J., Roussopoulos, M., Welsh, M., and Seltzer, M., "Network-aware operator placement for stream-processing systems," in *Proceedings of the 22nd International Conference on Data Engineering*, ICDE '06, (Washington, DC, USA), pp. 49–, IEEE Computer Society, 2006.

[114] Plimpton, S., "Fast parallel algorithms for short-range molecular dynamics," *J. Comput. Phys.*, vol. 117, pp. 1–19, Mar. 1995.

[115] Polte, M., Simsa, J., Tantisiriroj, W., and Gibson, G., "Fast log-based concurrent writing of checkpoints," in *In Proceedings of Petascale Data Storage Workshop 2008 at Supercomputing 2008*, 2008.

[116] Pritchard, H., Gorodetsky, I., and Buntinas, D., "A ugni-based mpich2 nemesis network module for the cray xe," in *Proceedings of the 18th European MPI Users' Group Conference on Recent Advances in the Message Passing Interface*, EuroMPI'11, (Berlin, Heidelberg), pp. 110–119, Springer-Verlag, 2011.

[117] Pugmire, D., Childs, H., Garth, C., Ahern, S., and Weber, G. H., "Scalable computation of streamlines on very large datasets," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, (New York, NY, USA), pp. 16:1–16:12, ACM, 2009.

[118] Rabenseifner, R., Hager, G., and Jost, G., "Hybrid mpi/openmp parallel programming on clusters of multi-core smp nodes," in *Proceedings of the 2009*

17th Euromicro International Conference on Parallel, Distributed and Network-based Processing, PDP '09, (Washington, DC, USA), pp. 427–436, IEEE Computer Society, 2009.

[119] Rajachandrasekar, R., Ouyang, X., Besseron, X., Meshram, V., and Panda, D. K., "Can checkpoint/restart mechanisms benefit from hierarchical data staging?," in *Euro-Par Workshops (2)*, pp. 312–321, 2011.

[120] Rübel, O., Prabhat, Wu, K., Childs, H., Meredith, J., Geddes, C. G. R., Cormier-Michel, E., Ahern, S., Weber, G. H., Messmer, P., Hagen, H., Hamann, B., and Bethel, E. W., "High performance multivariate visual data exploration for extremely large data," in *SC*, p. 51, 2008.

[121] Rübel, O., Prabhat, Wu, K., Childs, H., Meredith, J. S., Geddes, C. G. R., Cormier-Michel, E., Ahern, S., Weber, G. H., Messmer, P., Hagen, H., Hamann, B., and Bethel, E. W., "High performance multivariate visual data exploration for extremely large data," in *SC*, p. 51, 2008.

[122] Ryu, K. D. and Hollingsworth, J. K., "Linger longer: Fine-grain cycle stealing for networks of workstations," in *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*, Supercomputing '98, (Washington, DC, USA), pp. 1–12, IEEE Computer Society, 1998.

[123] Sandberg, A., Eklöv, D., and Hagersten, E., "Reducing cache pollution through detection and elimination of non-temporal memory accesses," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, (Washington, DC, USA), pp. 1–11, IEEE Computer Society, 2010.

[124] Schlosser, S. W., Ryan, M. P., Taborda-Rios, R., López, J., O'Hallaron, D. R., and Bielak, J., "Materialized community ground models for large-scale earthquake simulation," in *SC*, p. 54, 2008.

[125] Schnitzer, B. and Leutenegger, S. T., "Master-client r-trees: A new parallel r-tree architecture," in *Proceedings of the 11th International Conference on Scientific and Statistical Database Management*, SSDBM '99, (Washington, DC, USA), pp. 68–, IEEE Computer Society, 1999.

[126] Seamons, K. E., Chen, Y., Jones, P., Jozwiak, J., and Winslett, M., "Server-directed collective i/o in panda," in *Supercomputing '95: Proceedings of the 1995 ACM/IEEE conference on Supercomputing (CDROM)*, (New York, NY, USA), p. 57, ACM, 1995.

[127] Shalf, J., Dosanjh, S. S., and Morrison, J., "Exascale computing technology challenges," in *VECPAR*, pp. 1–25, 2010.

[128] Shekhar, R., Fayyad, E., Yagel, R., and Cornhill, J. F., "Octree-based decimation of marching cubes surfaces," in *Proceedings of the 7th Conference*

on Visualization '96, VIS '96, (Los Alamitos, CA, USA), pp. 335–ff., IEEE Computer Society Press, 1996.

[129] Singh, A., Balaji, P., and Feng, W.-c., "Gepsea: A general-purpose software acceleration framework for lightweight task offloading," in *Proceedings of the 2009 International Conference on Parallel Processing*, ICPP '09, pp. 261–268, 2009.

[130] Sinha, R. R. and Winslett, M., "Multi-resolution bitmap indexes for scientific data," *ACM Trans. Database Syst.*, vol. 32, no. 3, p. 16, 2007.

[131] Son, S. W., Lang, S., Carns, P., Ross, R., Thakur, R., Ozisikyilmaz, B., Kumar, P., Liao, W.-K., and Choudhary, A., "Enabling active storage on parallel i/o software stacks," in *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, 2010.

[132] Stockinger, K., Shalf, J., Bethel, E. W., and Wu, K., "Dex: Increasing the capability of scientific data analysis pipelines by using efficient bitmap indices to accelerate scientific visualization," in *SSDBM*, pp. 35–44, 2005.

[133] Sun, Y., Zheng, G., Kale, L. V., Jones, T. R., and Olson, R., "A ugni-based asynchronous message-driven runtime system for cray supercomputers with gemini interconnect," in *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium*, IPDPS '12, (Washington, DC, USA), pp. 751–762, IEEE Computer Society, 2012.

[134] Tang, L., Mars, J., and Soffa, M. L., "Compiling for niceness: Mitigating contention for qos in warehouse scale computers," in *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, (New York, NY, USA), pp. 1–12, ACM, 2012.

[135] Tang, L., Mars, J., Wang, W., Dey, T., and Soffa, M. L., "Reqos: Reactive static/dynamic compilation for qos in warehouse scale computers," in *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, (New York, NY, USA), pp. 89–100, ACM, 2013.

[136] Tu, T., Rendleman, C. A., Borhani, D. W., Dror, R. O., Gullingsrud, J., Jensen, M. Ø., Klepeis, J. L., Maragakis, P., Miller, P., Stafford, K. A., and Shaw, D. E., "A scalable parallel framework for analyzing terascale molecular dynamics simulation trajectories," in *SC*, p. 56, 2008.

[137] Tu, T., Yu, H., Bielak, J., Ghattas, O., López, J. C., Ma, K.-L., O'Hallaron, D. R., Ramirez-Guzman, L., Stone, N., Taborda-Rios, R., and Urbanic, J., "Analytics challenge - remote runtime steering of integrated terascale simulation and visualization," in *SC*, p. 297, 2006.

[138] Tu, T., Yu, H., Ramirez-Guzman, L., Bielak, J., Ghattas, O., Ma, K.-L., and O'Hallaron, D. R., "Scalable systems software - from mesh generation to scientific visualization: an end-to-end approach to parallel supercomputing," in *SC*, p. 91, 2006.

[139] W. X. Wang, Z. Lin, W. M. T. W. W. L. S. E. J. L. V. L. G. R. T. S. H. and Manickam, J., "Gyrokinetic simulation of global turbulent transport properties in tokamak experiments," *Physics of Plasma*, vol. 13, 2006.

[140] Wang, C., Vazhkudai, S. S., Ma, X., Meng, F., Kim, Y., and Engelmann, C., "Nvmalloc: Exposing an aggregate ssd store as a memory partition in extreme-scale machines," in *IPDPS*, pp. 957–968, 2012.

[141] Wang, C., Rayan, I. A., Eisenhauer, G., Schwan, K., Talwar, V., Wolf, M., and Huneycutt, C., "Vscope: Middleware for troubleshooting time-sensitive data center applications," in *Middleware*, pp. 121–141, 2012.

[142] Wang, C., Rayan, I. A., and Schwan, K., "Faster, larger, easier: Reining real-time big data processing in cloud," in *Proceedings of the Posters and Demo Track*, Middleware '12, (New York, NY, USA), pp. 4:1–4:2, ACM, 2012.

[143] Wang, C., Schwan, K., Talwar, V., Eisenhauer, G., Hu, L., and Wolf, M., "A flexible architecture integrating monitoring and analytics for managing large-scale data centers," in *ICAC*, pp. 141–150, 2011.

[144] Wang, H., Parthasarathy, S., Ghoting, A., Tatikonda, S., Buehrer, G., Kurç, T. M., and Saltz, J. H., "Design of a next generation sampling service for large scale data analysis applications," in *ICS*, pp. 91–100, 2005.

[145] Wang, Y., Jiang, W., and Agrawal, G., "Scimate: A novel mapreduce-like framework for multiple scientific data formats," in *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (Ccgrid 2012)*, CCGRID '12, (Washington, DC, USA), pp. 443–450, IEEE Computer Society, 2012.

[146] Welch, B., Unangst, M., Abbasi, Z., Gibson, G. A., Mueller, B., Small, J., Zelenka, J., and Zhou, B., "Scalable performance of the panasas parallel file system," in *FAST*, pp. 17–33, 2008.

[147] Wickremesinghe, R., Chase, J. S., and Vitter, J. S., "Distributed computing with load-managed active storage," in *In HPDC-11*, 2002.

[148] Wolf, M., Cai, Z., Huang, W., and Schwan, K., "Smartpointers: personalized scientific data portals in your hand," in *SC*, pp. 1–16, 2002.

[149] Yang, Q. and Ren, J., "I-cash: Intelligently coupled array of ssd and hdd," in *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture*, HPCA '11, (Washington, DC, USA), pp. 278–289, IEEE Computer Society, 2011.

[150] Yu, H. and Ma, K.-L., "A study of i/o methods for parallel visualization of large-scale data," *Parallel Comput.*, vol. 31, no. 2, pp. 167–183, 2005.

[151] Yu, H., Wang, C., Grout, R. W., Chen, J. H., and Ma, K.-L., "In situ visualization for large-scale combustion simulations," *IEEE Computer Graphics and Applications*, vol. 30, no. 3, pp. 45–57, 2010.

[152] Yu, H., Wang, C., Grout, R. W., Chen, J. H., and Ma, K.-L., "In situ visualization for large-scale combustion simulations," *IEEE Computer Graphics and Applications*, vol. 30, no. 3, pp. 45–57, 2010.

[153] Yu, H., Wang, C., and Ma, K.-L., "Massively parallel volume rendering using 2-3 swap image compositing," in *SC*, p. 48, 2008.

[154] Yu, W., Vetter, J. S., and Oral, S., "Performance characterization and optimization of parallel i/o on the cray xt," in *IPDPS*, pp. 1–11, 2008.

[155] Zhang, F., Docan, C., Parashar, M., and Klasky, S., "Enabling multi-physics coupled simulations within the pgas programming framework," in *CCGRID*, pp. 84–93, 2011.

[156] Zhang, F., Docan, C., Parashar, M., Klasky, S., Podhorszki, N., and Abbasi, H., "Enabling in-situ execution of coupled scientific workflow on multi-core platform," in *IPDPS*, pp. 1352–1363, 2012.

[157] Zhang, L. and Parashar, M., "Seine: a dynamic geometry-based shared-space interaction framework for parallel scientific applications," *Concurrency and Computation: Practice and Experience*, vol. 18, no. 15, pp. 1951–1973, 2006.

[158] Zhang, X., Jiang, S., and Davis, K., "Making resonance a common case: A high-performance implementation of collective i/o on parallel file systems," in *IPDPS*, 2009.

[159] Zheng, F., Abbasi, H., Cao, J., Dayal, J., Schwan, K., Wolf, M., Klasky, S., and Podhorszki, N., "In-situ i/o processing: A case for location flexibility," in *Proceedings of the Sixth Workshop on Parallel Data Storage*, PDSW '11, (New York, NY, USA), pp. 37–42, ACM, 2011.

[160] Zheng, F., Abbasi, H., Docan, C., Lofstead, J. F., Liu, Q., Klasky, S., Parashar, M., Podhorszki, N., Schwan, K., and Wolf, M., "Predata - preparatory data analytics on peta-scale machines," in *IPDPS*, pp. 1–12, 2010.

[161] Zheng, F., Venkatramani, C., Wagle, R., and Schwan, K., "Cache topology aware mapping of stream processing applications onto cmps," in *Proceedings of the 2013 IEEE 33rd International Conference on Distributed Computing Systems*, ICDCS '13, (Washington, DC, USA), pp. 52–61, IEEE Computer Society, 2013.

[162] Zheng, F., Zou, H., Eisenhauer, G., Schwan, K., Wolf, M., Dayal, J., Nguyen, T.-A., Cao, J., Abbasi, H., Klasky, S., Podhorszki, N., and Yu, H., "Flexio: I/o middleware for location-flexible scientific data analytics," in *IPDPS*, pp. 320–331, 2013.

[163] Zhuravlev, S., Blagodurov, S., and Fedorova, A., "Addressing shared resource contention in multicore processors via scheduling," in *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, (New York, NY, USA), pp. 129–142, ACM, 2010.

VITA

Fang Zheng was born in Yongzhou, Hunan Province in China. He received his Bachelor and Master degrees in Computer Science and Technology from Xi'an Jiaotong University in China, in 2004 and 2007, respectively. He started his Ph.D. study in Computer Science in College of Computing, Georgia Institute of Technology since August 2007. He has been working with his advisor Professor Karsten Schwan in the CERCS research center and collaborating with scientists from Oak Ridge National Laboratory and Sandia National Laboratories. His research is focused on addressing the data management challenges for large scale scientific applications, and developing I/O middleware solutions to enable high performance I/O and scalable online scientific analytics. His work has been applied to several leadership scientific applications and demonstrated notable performance improvements for those applications. He also collaborated with researchers in IBM T.J. Watson Research Center on optimizing stream processing applications on multi-core platforms.