MIDDLEWARE FOR LARGE SCALE IN SITU ANALYTICS WORKFLOWS

A Thesis
Presented to
The Academic Faculty

by

Jai Dayal

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

Georgia Institute of Technology
December 2016

MIDDLEWARE FOR LARGE SCALE IN SITU ANALYTICS WORKFLOWS

Approved by:

Dr. Matthew Wolf, Committee Chair
Computer Science and Mathematics Division
Oak Ridge National Laboratory

Professor Ada Gavrilovska
School of Computer Science
Georgia Institute of Technology

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

Professor Santosh Pande
School of Computer Science
Georgia Institute of Technology

Professor Ling Liu
School of Computer Science
Georgia Institute of Technology

Dr. Gerald Lofstead
Computer Science Research Institute
Sandia National Laboratories

Date Approved: October 28, 2016

To my wife, parents, and in-laws.

ACKNOWLEDGEMENTS

First and foremost, I would like to thank my advisors, Professor Karsten Schwan and Dr. Matthew Wolf. I am honored to be their student and their patience, knowledge, and pedagogical skills will serve as guiding principals for the rest of my career. Karsten's loss was devastating for all of us, but without Matt's dedication to the students, my dissertation would have not been possible.

I'd like to give thanks to Greg Eisenhauer for his endless moral and technical support over the last years. It was nice to know that if I had to pull an all nighter, he'd be willing to as well. Similarly, I would like to give a special thanks to Jay Lofstead for the years of collaboration, internship opportunities, invaluable advice as well as being an excellent host whenever I am in Albuquerque.

My fellow students and friends, Alex Merritt, Anshuman Goswami, and Hasan Abbasi all deserve thanks as they provided numerous hours of entertainment as well as technical and research advice over the years. My time at Georgia Tech was greatly enjoyable because of their friendship.

Additionally, I'd like to thank Professors Ling Liu, Ada Gavrilovska, and Santosh Pande for being on my thesis committee and providing stability during unstable times with Karsten's passing.

Susie McClain deserves high praise for all of her efforts on making sure all the important details are taken care of and always providing assistance to me when needed.

Finally, I'd like to thank my wife and family for their endless support and patience as they often have had to put their lives on hold while I navigated my way through the program.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

The trend to exascale is causing researchers to rethink the entire computational science stack, as future generation machines will contain both diverse hardware environments and run times that manage them. Additionally, the science applications themselves are stepping away from the traditional bulk-synchronous model and are moving towards a more dynamic and decoupled environment where analysis routines are run in situ alongside the large scale simulations.

This thesis presents CoApps, a middleware that allows in situ science analytics applications to operate in a location-flexible manner. Additionally, CoApps explores methods to extract information from, and issue management operations to, lower level run times that are managing the diverse hardware expected to be found on next generation exascale machines. This work leverages experience with several extremely scalable applications in materials and fusion, and has been evaluated on machines ranging from local Linux clusters to the supercomputer Titan.

CHAPTER I

INTRODUCTION

1.1 Trend: In Situ Analytics and Hardware Heterogeneity

On current generation petascale supercomputers, large-scale science applications like GTC [2] and S3D [34] are already stressing the limits of the capabilities these machines provide. By offering diverse hardware architectures, such as inter-mixed CPUs and accelerators, deep memory hierarchies, and multi-tiered storage systems, users can now scale their science applications to conduct science at larger scales and finer resolutions. Scaling to petascale sizes can be problematic as data produced at such large volumes and high velocities can overwhelm storage systems leading to significant performance bottlenecks. As we move beyond petascale towards exascale infrastructures, this trend only intensifies.

While these immense scales and data volumes have given science end users better insights into the phenomena they are investigating, before such insights can be gleaned, the data must often go through an expensive and complex analysis and visualization process. The desire to scale analytics workflows to handle such data volumes, and to better take advantage of the new hardware found on modern machines, has caused researchers to investigate new methods for conducting computational science to help avoid the existing scalability bottlenecks. In particular, science and analysis workflows are moving away from relying solely on storage systems as the intermediary for the data, to instead using models where analysis workflows run concurrently, or "in situ" with the core simulation ingesting data live as it is being produced. In situ is a Latin phrase used in a number of science and engineer fields that means "on site" or "in place." In computational science, in situ analysis means to analyze the data

in the same location as it is produced, which has been taken to more generally mean the data is analyzed before it is sent to storage. An open question for science end users then is what is the proper location to execute this analysis code in relation to the simulation: (1) as separate processes on the same set of compute nodes as the simulation, either sharing cores or on dedicated cores; (2) as separate processes on a set of dedicated "staging" nodes; (3) being treated as standard in-line functions that share the simulation's address space; and (4) as traditional post-run analysis using storage as the medium.

Beyond addressing storage-related performance challenges, in situ analytics offer science users **new functionality** for better understanding the scientific simulations being run. This includes using analytics to (1) continuously ascertaining simulation validity, (2) gaining rapid insights into the scientific processes being simulated (online visualization), (3) managing ensembles of simulations, or even (4) enabling methods for application steering.

Additionally, several programming models [12, 60], run times [19, 67, 33, 26], and operating systems [32, 54] have been developed to provide science end users with better ways to develop their analytics to take advantage of the growing number of hardware components. Figure 1 depicts an example of what a supercomputer node is projected to look like at exascale. The individual components of this example are not particularly relevant for this thesis, but what should be noted is the large variety of hardware and software subsystems that end users will have at their diposal on an exascale machine.

While the research supporting these developments is already showing great promise, there are still several challenges in terms of "bridging the gap" between the workflow components and the underlying hardware resources. Analytics algorithms all have different scaling characteristics, resilience properties, and any given algorithm can also have different versions targeted for specific hardware (i.e., a histogram function can

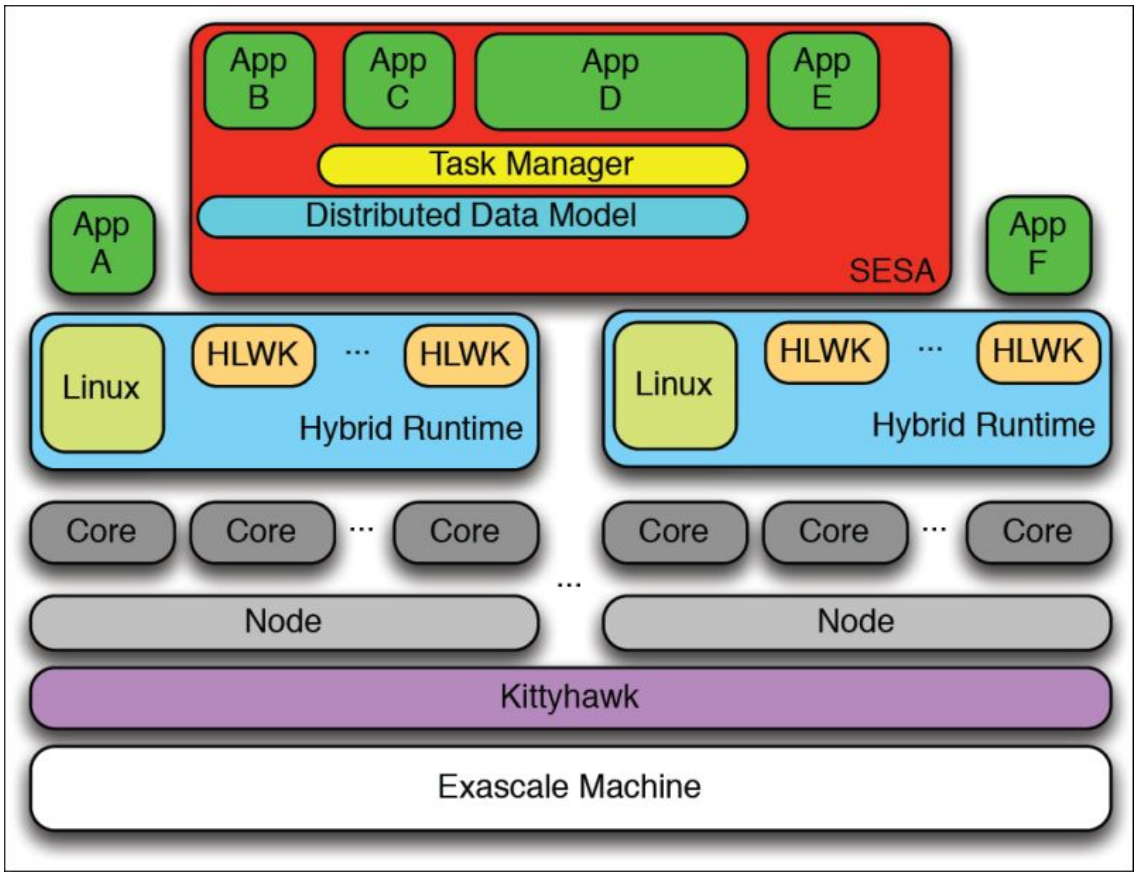


Figure 1: Projected Node Architecture for an Exascale Machine

have a CPU or a GPU implementation). Relying on the end-user to properly profile and schedule the workflow components on the different hardware is time consuming and error prone, only to be exacerbated at exascale.

Further, dynamics can also occur at run time causing initial resource allocations to become sub-optimal. Such dynamism can occur for several reasons; poor initial resource allocation to the workflow components leading to performance bottlenecks, hardware and software failures, the nature of the changing data may change the run-time, etc. At the current state of the art, science end users over-provision for worst case scenarios leading to wasted resources.

The focus of this thesis is to design middleware constructs for in situ analytics workflows that enables proactive and reactive response to run time dynamics in order to meet user-driven performance and science goals. Specifically, I am presenting CoApps and its enabling technical components that will provide: (1) communication mechanisms so that workflow components can operate in a location independent manner; (2) an execution mechanism that can make use of different implementations of analytics algorithms to enable smarter placement decisions; (3) provide a programmatic API to allow users to create beyond simple, best effort service level agreements (SLAs) that the middleware can enforce at run time.

The communication mechanisms address both inter-component and intra-component needs, the latter stemming from colocation limitations found on many high-end machines. The execution framework chooses at run time from the different hardware implementations of the analytics components in an effort to improve consolidation opportunities and better meet end-user goals. The programming API allows information about the state of the workflow to be gathered and evaluated and allows the end-user to specify control operations to take when certain conditions or triggers are met.

The CoApps approach rests on assumptions that hold true for many large-scale

scientific applications and their associated analytics workflows. These assumptions do not always match those found in enterprise or "big data" frameworks like [64, 6, 3] in terms of their data characteristics, execution models, and degrees of parallelism.

- **Functional Dependencies.** Analytics codes expect to ingest data matching specific formats and layouts. These analytics functions may need to transform data to meet algorithmic correctness and/or to export an analysis function's discoveries into the data itself. Given these dependencies in the data-plane, functions in an analytics pipeline may or may not require in-order operation.

- **Heterogeneous Codes.** Analytics can have heterogeneous architectures and have a wide range of execution models, fault tolerance, and scaling characteristics.

- **Stringent Resource Constraints.** Resources are typically assigned to the compute job statically. Analysis codes are given "spare" resources, i.e., spare CPU cycles on simulation nodes [67, 14], reserved staging nodes [9, 25], or those on smaller, auxiliary clusters perhaps in different physical locations. Analytics pipelines, therefore, must operate with these limited resources, without interfering with the simulations and their output actions including by delaying simulation completion or adversely affecting other jobs running on the same platforms.

1.2 Thesis Statement

In situ analytics provide a high performance path for modern scientific applications and workflows running on leadership class machines. For such applications, enabling in situ analytics to be proactive and reactive to runtime dynamics creates inherently scalable workflows that can be managed at runtime to better utilize machine resources, improve application performance, and to better meet end-user goals.

1.3 Thesis Contributions

There are four principle contributions of this thesis. First, this thesis introduces the CoApps abstraction, which enables colocation of workflow components. Second, this thesis introduces communication mechanisms that enable location independence as well as the reorganization of in situ workflow components at run time. These techniques include both inter- and intra-component messaging and data exchanges as well as the co-management of network resources. Third, this thesis provides an execution environment to make use of different hardware implementations of workflow components when making decisions on colocation. Finally, we explore these abstractions and mechanisms using real science codes operating on current high-end petascale supercomputers as well as smaller university scale clusters.

1.4 Impact of Future Technologies

This thesis provides a software, and more specifically a middleware, for the run time management of in situ science analytics as we approach exascale limits. Hardware, and the lower-level run times that management, also provide solutions for improving the scalability and programmability of science workflows. New technologies such as burst buffers and non-volatile memory (NVRAMS) are seeking to address the I/O challenges at exascale, however these approach are complimentary to the approach presented in this thesis. In particular fast persistent storage hardware can improve our resilience and data exchange capabilities. This is discussed further in section 7.

1.5 CoApps Research Overview

The remainder of this thesis presents a progressive narrative of the requirements and performance of the CoApps system, starting with some canonical science applications, through our development of in situ workflows to the CoApps abstraction.

Chapter 2 outlines several driving applications that largely served as motivating

use cases for CoApps, and in situ workflows in general. There, we present science applications, their analysis workflows and some of the unique challenges they represent for the CoApps abstraction. We also present some of the core technologies we use as part of our CoApps implementation. Chapter 3 describes our work on leveraging a publish/subscribe infrastructure to design a communication, or code-coupling, mechanism for "on line" science workflows. Chapter 4 builds on this initial concept and describes abstractions and control protocols for orchestration. This work focuses on coarse grained policies mostly with re-arranging workflow components in a staging area and does not directly deal with per compute node attributes. Chapter 5 presents the complete CoApps abstraction and run time. It describes how location independence is achievable at run time, presents new launching and communication mechanisms for node-sharing in situ cases, and also describes what additional information and control structures are needed to make more fine-grained per compute node decisions at run time. Chapter 7 discusses future work, Chapter 6 discusses related work on in situ workflows and orchestration/management in general, and Chapter 8 concludes the thesis.

CHAPTER II

MOTIVATING APPLICATIONS AND FOUNDATIONAL TECHNOLOGIES

CoApps provides a new paradigm for programming, deploying and executing in situ scientific workflows on leadership scale applications. The design principles have been developed in collaboration with scientific application developers, and with the scientists using these applications. We will look at some of these motivating application, and also describe the workflow management challenges raised by each. The CoApps abstraction can be considered to be one way of addressing these challenges.

2.1 Motivating Applications

2.1.1 LAMMPS

LAMMPS[53] is a molecular dynamics simulation used across a number of science domains ranging from materials engineering to physics to biology. Depending on the particular input parameters used with it, it can have a wide variety of performance characteristics. It is written with MPI and performs force and energy calculations on discrete atomic particles. After a number of user-defined epochs, it outputs the atomistic simulation data (e.g., atom types and positions) with the size of this data ranging fom megabytes to terabytes depending on the science being investigated. LAMMPS can also take advantage of common hardware runtimes such as CUDA and OpenMP and other extensions are available to use newer accelerator based technologies such as Xeon Phi.

2.1.1.1 SmartPointer

SmartPointer[61] is a representative analytics pipeline interpreting LAMMPS output data to detect and then scientifically explore plastic deformation and crack genesis. In such scenarios, the material being simulated is steadily stressed until it first starts to break. The scientific question being asked is how to understand the geometry of the region around that initial break. This means that the purpose of the molecular dynamics simulation is to bring the data set to some self-consistent, interesting state, at which point substantial additional analytics and characterization need to take place. The SmartPointer analytics toolkit implements these functions to determine where and when plastic deformation occurs and to generate relevant information as the material is cracked. Table 1 summarizes the computational characteristics of the individual SmartPointer components, and the list below explains each in greater detail. Figure 2 depicts this example workflow.

This workflow provides us with several interesting orchestration challenges as it requires application introspection into the data based on the CSYM and CNA components. In contrast to purely performance based policies, the analysis functions report the metric of interest (CSYM detects a crack) and the orchestration actions (kill CSYM and run CNA) are defined by and triggered by the applications. This type of data-centric policy ensures data quality via correct execution of pipeline analysis functions.

- *Lammps Helper*: serves as an aggregator and filter of the raw LAMMPS data.
- *Bonds*: subscribes to aggregated data from Lammps helper at each time-step, and performs an all-nearest neighbor calculation to determine which atoms are bonded together in order to publish a bond-pair array as its output.
- *Csym*: the central symmetry analysis routine operates on an array of bond-pairs from bonds, and also a bond adjacency list (a graph structure) to determine if

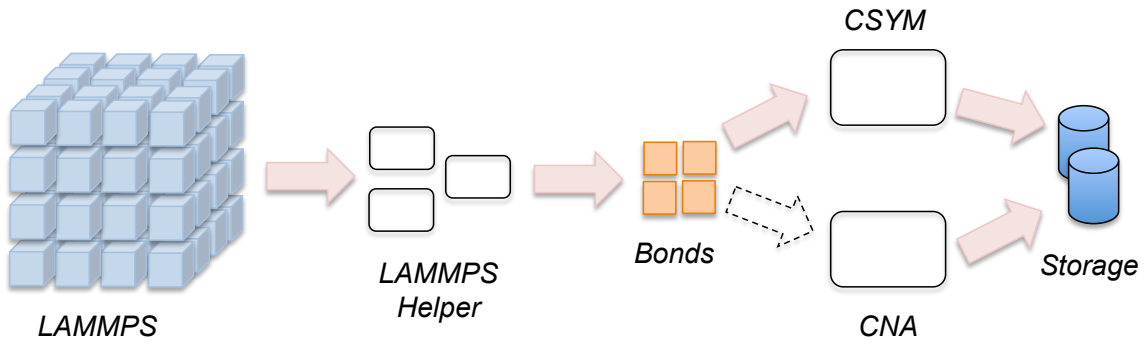


Figure 2: LAMMPS and SmartPointer Analysis Pipeline

Table 1: Characteristics of SmartPointer Analysis Actions

	Complexity	Data Model	Stateful
Helper	$O(n)$	Array	No
Bonds	$O(n^2)$	Array, Parallel	No
CSym	$O(n)$	Complex	Yes
CNA	$O(n^3)$	Array	No

there is a deformation in the material. This code maintains the initial bonds adjacency state for the duration of its run.

- *CNA*: common neighbor analysis executes whenever CSYM determines that a deformation in the material has occurred. CNA is compute-intensive and is executed on the bond pairs array to perform a structural characterization on the data, to determine the conditions under which the crack occurred.

2.1.2 GTCP

GTCP[44], simulates a toroidally-confined plasma such as is found in Tokamak fusion reactors. In order to simulate such reactor environments, the code uses a Particle-in-Cell technique, where the electric and magnetic fields within the domain are stored on a mesh, but the ions of the plasma are represented by discrete particles. There are challenges with analyzing both the particle and mesh-carried variables, but for the purposes of this work, we have focused on the mesh ones. As such, the output of the simulation is a 3D array with the dimensions representing: (a) toroidal ranks

(toroidal slice number), (b) grid point numbers, and (c) property indices (ie magnetic field components, pressure, temperature). An example workflow that has motivated much of our development involves generating a per-timestep histogram that shows the distribution of per-gridpoint parallel pressure across the entire simulation. From GTCPs output, the particular quantities of interest must be extracted and then a histogram generated.

2.1.3 SuperGlue Reusable Analysis Workflow

SuperGlue[8] is a set of "off the self" analysis tools that are designed to be generic enough to be reusable for a large variety of different workflows. There are a number of analysis routines that can be chained together to form arbitrary workflows and users use configuration files to customize the workflow to the specific application. SuperGlue components are coupled together using the ADIOS framework, discussed in Chapter 2.2.1.

From the larger number of SuperGlue components available, we use the following:

- *Select*: Given an input stream that includes an n-dimensional array, Select extracts specific indices from one of the dimensions. The output array of this function has the same number of dimensions, but has a smaller number of elements.

- *Dim-Reduce*: removes one dimension from its input array by absorbing it into another dimension without modifying the total size of the data. The other dimensions are left unchanged.

- *Magnitude*: calculates the magnitudes of the vectors from the values of their individual components and outputs a 1D array of the new values.

- *Histogram*: Computes a histogram over the elements in the array. The histogram component is implemented using the Thrust Parallel Algorithms Library

(cite thrust) and runs on CPUs as well as accelerators such as GPUs.

The interesting part of this workflow, from an orchestration perspective, is that some of the components, i.e., the histogram component, have both CPU and GPU implementations available, giving us greater options to explore improving utilization hetergenous for hetergenous hardware. With current state of the art tools and deployment methods, end users are required to provision the workflow resources accordingly and often may choose sub-optimal arrangements. By offering different hardware implementations of workflow components, we have a greater number of placement options available to try to improve our ability to meet end-user goals.

2.2 Foundational Technologies

While CoApps is a significant change in the paradigm of workflow and data management for high performance applications, their development has been realized through the use of existing technologies that provide key characteristics necessary for the CoApps paradigm. Two components, ADIOS and EVPath, have played a significant roll in the implementation for the CoApps abstraction.

2.2.1 ADIOS

The Adaptable IO System (ADIOS) is an I/O componentization library that exposes file-like read and write interfaces to applications, with underlying I/O methods including disk based methods like POSIX and MPI-IO, and "staging" methods like Datatap[9], Dataspaces[25], and also Flexpath[22], an enabling technology for CoApps, which is further described in Chapter 3. With ADIOS, end users can simply 'switch' transports, without modifying their codes, using an external XML document. This allows us to identify those transports as well as other I/O characteristics, like the variables to be written, their array dimensions & offsets, etc. We have chosen ADIOS to be the interface into Flexpath for two reasons: 1) For ease of use by the

large number of existing applications that use the ADIOS interfaces, and 2) it provides a service-oriented interface for science applications allowing them to be written generically and deployed anywhere without extensive code changes or re-compilation efforts.

2.2.2 EVPath

EVPath [28] is a constructor for typed messaging systems. Using EVPath allows us to construct a number of underlying messaging infrastructures needed to realize the CoApp inplementation. Using EVpath, we have created type-based publish/subscribe messaging systems, described in Chapter 3, monitoring and control overlays described in Chapter 4.3.3, as well as transactional and RPC style messaging systems. Additionally, EVPath supports dynamic code generation techniques that allow us to deploy at runtime customized operations into the various messaging systems to give applications greater control over their data streams. This research has caused much fruitful back-and-forth with the core EVPath development process.

CHAPTER III

FLEXPATH: TYPE-BASED PUBLISH SUBSCRIBE FOR HIGH-END IN SITU WORKFLOWS

3.1 Introduction

This chapter presents Flexpath, a *type-based* publish/subscribe infrastructure for coupling high-end scientific applications with their online analytics services. The work presented here serves as our initial look at in situ workflow construction and addresses the need for data exchanges across parallel workflow components. While the initial implementation presented here focused on "in transit" workflows, we were able to extend the implementation for in situ workflows, as described in chapter 5.

Flexpath's pub/sub approach makes possible runtime configurability, scalability, and also fault tolerance, as the pub/sub abstraction allows for the decoupling of diverse analytics components, permits multiple subscribers or publishers to share a single data stream, and suppresses communications for cases in which there are no subscribers to certain data streams (e.g., those not of current interest). This is particularly well suited for the in situ analytics approach, as the core simulation may therefore be structured to make available a substantial array of internal data, knowing that only those parts that are needed at runtime will actually be exported. These properties contrast with the typical assumptions made by communication infrastructures like MPI, where the domain of executing processes is initialized at launch and cannot grow or shrink for the remainder of the execution.

With Flexpath, one can construct and dynamically manage or change the data processing pipelines or workflows needed for runtime analysis of the large volumes of this output data in ways that meet the following four design requirements of these

sorts of applications: (1) decouple analytics services from simulation codes, (2) maintain levels of performance similar to those obtained by analytics routines statically embedded with simulations, (3) permit those pipelines to cross node and/or machine boundaries, and (4) support the creation of higher level methods for managing these pipelines. Sample management constructs built in our own previous work [23], for example, have balanced pipeline operations to ensure QoS and have implemented transactional constructs with the goal of providing ACID properties for select online analytics [62, 46].

Flexpath's pub/sub communication mechanism, key to meeting design objective (1), obtains flexibility for component-component communications, without the performance penalties incurred by traditional broker-based pub/sub infrastructures. This technical contribution is achieved by using direct connections between interacting components, including the scatter-gather or MxN communications needed across different communicating internally parallelized analytics components. This high performance implementation for such peer-to-peer techniques utilizes a subscription implementation, allowing readers to specify derived versions of messages, e.g., to receive only those slices of data objects they require, as well as registering dynamic transformations of typed objects when there are mismatches between publishers and subscribers, e.g. row to column order array conversions.

With regards to the need to maintain performance in cross-platform environments (design objectives 2 and 3), Flexpath has been built to leverage multiple underlying communication protocols, ranging from a shared memory protocol employed for on-node communications, to the RDMA-based protocols existing on high end machines, to the TCP/IP protocols required for linking remote collaborators. As is described in Section 3.3, much of this comes from inheriting a multi-modal connection management system through the EVPath framework. Finally, with regards to design object (4)'s concerns for management, Flexpath's approach allows for it to export monitoring

data and management 'hooks' with which higher level management methods can be realized. As will be seen later, we utilize some simple workflow-level management schemas in this work, but future work will extend the complexity and robustness of this feature of the system.

Conceptually, Flexpath's development builds on extensive prior work on efficient parallel I/O pipelines, including data staging methods for running analytics and visualization [9, 66, 25, 58], data streaming and the online QoS control of such data streams [28, 51, 62], the aggressive use of source-based data reduction and filtering [39, 66], and convenient ways to carry out remote data visualization [61, 20, 18]. For high end machines, challenges include dealing with network congestion [9], providing data reliability when operating at scale [46], making data "right" for use by successive analytics codes without unnecessary data movement [10, 66], and dealing with application dynamics, as when codes are dynamically activated or de-activated. Such dynamics, in fact, have given rise to interesting methods used by modern data visualization systems like VisIt contracts [21].

Driven by such prior work, Flexpath is designed as a communication substrate that does not proscribe specific management methods. Instead, it makes possible the efficient realization of alternative communication scheduling techniques [9] and/or higher level methods for managing entire analytics workflows [23]. In contrast to web-based or commercial data streaming infrastructures [51, 40, 63], it does not constrain end users in how to write their analytics routines, so that they can leverage the rich tools already existing for these purposes, like R or MatLab. Finally, leveraging the ADIOS I/O APIs already in common use on petascale machines [45], Flexpath's implementation as an ADIOS 'transport method' allows it to adopt and adapt many off-line analytics pipelines that were originally structured as sets of independently programmable and deployable analytics services using ADIOS as the interface of choice [45].

Flexpath is deployed for use across a range of high end machines, including ORNL's Titan machine, Infiniband clusters, and commodity scientific computing engines. This chapter experimentally evaluates its technical elements and approach with two representative applications with significant scientific user communities, LAMMPS [53] and GTS [2], coupled with their associated data analytics service flows. For this chapter, experiments are run on Oak Ridge National Lab's Sith machine and on smaller-scale Linux clusters available at our own institution. Please see Chapter 2.1.1 for a discussion on LAMMPS and Smartpointer.

GTS is a plasma fusion simulation with an implementation that exploits coarse grained process level parallelism using MPI, and more fine-grained thread-level parallelism using OpenMP [2]. This particle-in-cell code has different output frequencies for both particles and mesh-level statistics. In order to examine the dynamics involved, in particular dangerous transient effects that might damage a real reactor vessel, it is useful to dynamically evaluate and characterize particular trends on the inner and outer edges of the plasma. Unlike the LAMMPS case, these transients are not as algorithmically identifiable, so secondary analysis methods are used to infer their existence, and then, much more detailed inspection involving direct interaction with the physicists is used to further the investigation. The GTS analytics pipeline used in this chapter computes parallel histograms of multiple grid-carried variables, and runs parallel-coordinate visualizations to provide suitable data to those physicists.

The LAMMPS and GTS analytics workflows have some important shared characteristics. In both examples, analytics codes are run as independent services, each simply executing its functions on the data that is available. The operation of the individual analysis routines are not affected by each other and generally, the codes are unaware of the details of how or when the other codes are run. This results in analytics workflows best described as sets of analytics services loosely coupled in terms of space, time, and synchronization.

3.2 Type-based Publish Subscribe for HPC Environments

Type-based publish/subscribe [29] is a pub/sub paradigm in which producers publish objects classified by type to a communication substrate, and consumers subscribe to them by specifying the types of objects in which they are interested. Here, type reflects both the structure of the published data as well as metadata extensions that can be determined at runtime. This distinction, along with other technical contributions, is part of what allows Flexpath to adopt a high performance direct-connect, rather than brokered, infrastructure.

While Flexpath uses a type-based pub/sub model, end users are not required to change their ADIOS codes or applications to adopt this new model. Instead, the Flexpath implementation exploits the model's several similarities with the standard file I/O model already known to science users. In the file I/O model, science applications exchange data by using a shared filesystem as the data exchange medium [48], and workflows are constructed through the use of intermediate files stored on disk. As a familiar scientific scenario, consider the following: at each output epoch the parallel writers open a file, encode their data in the proper metadata-rich serialization format, like NetCDF [42] or HDF5 [56], populate the write buffers, and finally, flush them to disk. Similarly, for a given read epoch the (possibly parallel) readers open the file, read metadata about the objects present in the file, create the appropriate buffers, perform the reads, and then seek ahead to the next block of data, if available. Additional attractive elements of the file-based approach include the ability to perform "seeks" to retrieve fine-grained slices of the available data, data durability and persistence guarantees.

A detraction from this model is the synchronous nature of file-based I/O and its poor performance at large scales. Particularly, if there is a complex trade-off between number of files, number of writers, and the layout of data within those files to be most scientifically useful, the attempt to optimize any one file system parameter

can yield sub-optimal results for the other. In the dynamic scientific investigations targeted by this work, a mis-predicted optimization could have profound impact on the viability of the runtime analysis if it were to use a traditional filesystem-based approach. A key reason for introducing our asynchronous, type-based pub/sub has been to avoid this disk bottleneck when linking simulations with dynamic sets of analytics services[66, 25, 9].

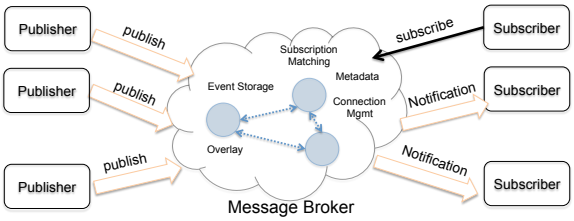
The properties of a type-based pub/sub system, although superficially quite different from file I/O, map relatively well to the subset of such general functions that are offered by high performance I/O libraries such as HDF5 or ADIOS. For example, file names serve as the naming convention for establishing a pub/sub "channel" between coupled applications, so that writers and readers can be logically mapped between publishers and subscribers to a shared data set. A key realization about the high performance I/O abstraction is that, since data is already laid out for the I/O system utilizing many higher-level concepts of data structure (arrays, slabs, meshes, etc.), a seek is not an arbitrary binary offset within the file. Instead, it maps quite well to metadata subscriptions or type-based derivations within the scope of type-based pub/sub. For example, a seek to a particular slice of a global array can also be interpreted by the pub/sub as a parameterization of the peer-to-peer subscription parameters. This structural, rather than byte-level, addressing of data in the high performance space is key to aligning the two paradigms.

Beyond this mapping of file I/O to equivalent pub/sub actions, there are additional properties of the online analytics workflows targeted by Flexpath that make them well suited for the type-based pub/sub paradigm. The input and output types of each component in the workflow are well defined, giving rise to clean mappings to equivalent pub/sub type descriptions. Additionally, since the components of the workflow operate independently of each other, this favors an asynchronous communication model not subject to the issues with tightly synchronized data exchanges in

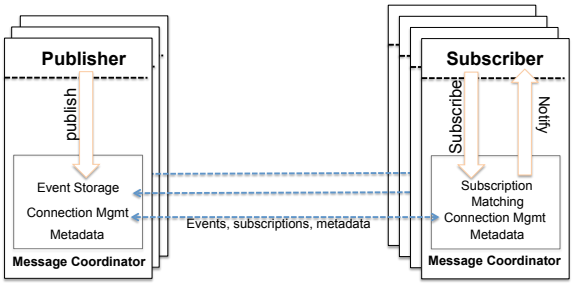
which senders block when downstream receivers are still processing the previous interval of data. This also supports the original design objective of being able to handle a heterogeneous computational environment. Finally, for analytics components that can change during the course of the run, key constituents of the types of workflows we aim to address, this amounts to dynamic changes in the data flow. Our pub/sub offers a model that allows such dynamics while not having to make the individual components aware of such run-time complexities.

While conceptually attractive, the efficient implementation of type-based pub/sub for high end applications and platforms poses significant challenges. To obtain high performance for large data volumes, we cannot use overlay routing techniques and/or move data to third-party brokers, as done in other traditional pub/sub implementations [16, 41, 38]. Second, unlike traditional type-based pub/sub, a single data object represented by a type is a collection of messages matching this type obtained from some large number of sources producing these messages, i.e., each process in the parallel application produces a portion of a global array as well as scalar variables that describe both global knowledge and the process's local view of the global array. Thus, the definition of type has to be extended to include notions of both local and global metadata parameters. Third, in the common MxN data exchanges that occur among science codes, a subscriber only wants to receive certain slices of the objects or in fact, objects transformed from one type to another. So, in addition to specifying types, subscribers also need to specifiy derivations on types. As a final complication, a type-based pub/sub infrastructure must have ways of dealing with type augmentations at runtime in order to be useful for adaptive codes like S3D.

Naturally, there are also machine-specific challenges to an efficient implementation of pub/sub on large-scale supercomputers. Structuring a solution which can both efficiently utilize highly specialized networking hardware and protocols, such as Infiniband and Cray's Gemini interconnect and simultaneously operate across multiple



(a) Traditional Pub/Sub



(b) Flexpath Pub/Sub

Figure 3: Traditional model of publish/subscribe vs. Flexpath model of publish/subscribe allowing for fine-grained data exchanges across parallel applications.

networking technologies, e.g. TCP/IP, in order to extend workflows across multiple machines and geographies, requires great care. There are issues of placement, throughput matching, and even security that must be addressed while maintaining both high performance and the simple pub/sub abstraction.

In summary, scientific simulations with complex online analytics workflows can benefit from the pub/sub paradigm, but given the non-trivial data exchange and I/O characteristics of these applications, and the nature of the systems on which they run, type-based publish/subscribe must be rethought to enable data exchanges at scale. Connection management, type derivations, and subscription management all must be re-addressed. We next explain the Flexpath architecture and implementation as it addresses these challenges.

3.3 Design and Implementation

Figure 3 depicts the conceptual design of Flexpath and contrasts it with the standard broker-based pub/sub model. In Flexpath, direct connections are established

between coordinators on opposing sides; when joining a channel, a subscriber selects a publisher peer coordinator it uses to retrieve publisher metadata. This metadata is used in conjunction with subscriber subscriptions to establish connections with publishers that own the requested data. Additionally, control messages are sent across connected coordinators to perform coordinated control operations like the eviction of expired data from the local data stores. We next describe some of the background needed for understanding the implementation of this functionality.

3.3.1 Background Technologies

3.3.1.1 EVPath Overview

The Flexpath messaging infrastructure is built on the EVPath [28] event-based transport middleware. EVPath supports the construction of active messaging overlay networks. User-defined data filtering and transformation functions reside in lightweight "stones" that serve as processing points in the overlay, and stones are linked to form overlay "paths", where the processes hosting these stones may reside on the same physical machine, on cluster nodes, or even on machines at different geographical locations. The filtering and transformation functions run by stones are implemented by registered call-back handlers written in C and statically associated with stones, or as inline functions deployed at runtime generated with the CoD (C-on-Demand) language. The types of EVPath stones used in the Flexpath implementation are the following:

- *Terminal Stone*: runs an application-registered call-back handler associated with an event type; the handler is invoked upon receipt of such an event.

- *Multi-Queue Stone*: operates over a collection of typed events, and allows users to implement policies like a tumbling window policy, or perform event transformations that span multiple event types.

- *Bridge Stone*: is used for network transmission, for communication with stones in a remote address space.

The additional stone types present in EVPath are described in [28].

Flexpath adopts from EVPath its methods for data serialization, termed Fast Flexible Serialization (FFS) [27], which means that Flexpath events are comprised of self-describing typed data elements, with types seen by all of the stones (and functions) operating on those events. The basic types supported are similar to those present in the C language, but with FFS, those types can be the building block for event data comprised of complex graph structures. We note that functions coded with CoD manipulating FFS encoded events can be generated at runtime and dynamically deployed to stones, in contrast with handlers that are compiled and deployed statically. Finally, to operate across several diverse communication protocols, Flexpath uses EVPath's networking abstraction, termed Connection Manager(CM), which currently supports as lower level protocols TCP/IP sockets, and via Sandia's NNTI [47], also high performance protocols like Infiniband, Cray's Gemini, and the Bluegene interconnect.

3.3.1.2 ADIOS Interface

The Adaptable IO System (ADIOS) is an I/O componentization library that exposes file-like read and write interfaces to applications, with underlying I/O methods including disk based methods like POSIX and MPI-IO, and "staging" methods like Datatap[9], Dataspaces[25], and also Flexpath. With ADIOS, end users can simply 'switch' transports, without modifying their codes, using an external XML document identifying those transports as well as other I/O characteristics, like the variables to be written, their array dimensions & offsets, etc. We have chosen ADIOS to be the interface into Flexpath for two reasons: 1) For ease of use by the large number of existing applications that use the ADIOS interfaces, and 2) as describe in section

3.2, there is a natural translation from ADIOS file-based I/O interfaces and type descriptions to Flexpath's pub/sub approach.

3.3.2 Implementation

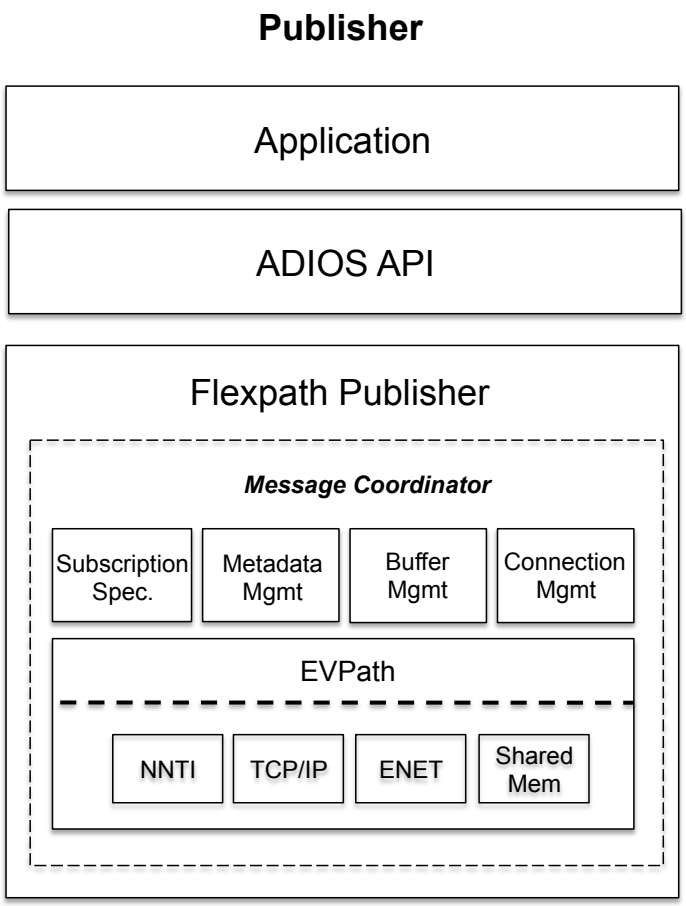


Figure 4: Software Architecture of Flexpath Publishers

Figure 4 depicts the software architecture of Flexpath from the publisher's perspective. The subscriber's interface is similar, except that it is layered beneath the ADIOS read interface.

Type Representation The publisher side of Flexpath obtains type information about data from the ADIOS data descriptor, generated by parsing the XML document during the *adios_open* call. This information is converted into a FFS format header

uniquely identified by a "cookie" transmitted along with the data. Upon the arrival of an event, if a receiver has not yet seen this cookie, the receiver issues a fetch request to the sender to obtain the FFS descriptor. This scheme avoids the redundant transmission of FFS metadata.

Publishing Data Publishers submit their data in Flexpath through the *adios_write* call, which is called for each variable to be written. Flexpath copies the data into the appropriate location in the FFS encoded buffer. This extra copy is not inherent to the pub/sub model, but is performed to satisfy the safety requirement of the ADIOS interface, which allows user codes to manage their own buffers. At the end of the output epoch, publishers perform a *publish* operation, available through the *adios_close* call, which submits the FFS encoded data to the local message coordinator. Additionally, on the publish operation, if there are any global arrays, we distribute each publisher's array offset metadata to all other publishers, so that the subscriber can ask its peer publisher coordinator for this information directly. This metadata allows us to extend the traditional definition of types to include local pieces of a larger global object.

Subscriptions Subscriptions are realized in three steps. First, the subscriber informs its local message coordinator about what variables and slices it needs. The message coordinator then fetches the global offset information for the given epoch from its peer writer coordinator and uses this information to determine from which publishers data is needed. The subscriber message coordinator will then send to each of those publisher coordinators a fetch message requesting the desired variables and slices. The offset metadata exchange also serves as our *notify* abstraction; metadata is only present for an output epoch if data for this epoch has been published.

In addition to the array slicing style subscriptions, we also allow for subscribers to specify type transformations, to allow publishers and subscribers to resolve type

mismatches. In the example listed in Chapter 2.1.1, the CSYM code actually wants to receive some data in the form of a bonds-adjacency list, a more complex graph structure, rather than only the bond-pair integer array published by the Bonds code. With Flexpath, this is done via transform operators, represented as CoD code or as a registered transform function. For this example, the transform function is run on the subscriber side to avoid having to transmit both sets of data.

Message Coordinators Message Coordinators are implemented by EVPath stones, CoD code, and with call-back handlers. On the publisher side, an EVPath multi-queue stone serves as an entry point for incoming messages and as the dispatcher for outgoing messages. Each publisher message coordinator maintains a local in-memory data buffer for storing published data as well as the associated metadata. It is this local data store, and separate threads for message processing and communication that allow Flexpath to have an asynchronous communication mode.. Additional functionality in message coordinators maintains reference counts to understand when data has been successfully received by all subscribers and perform subsequent data eviction operations, etc.

The subscriber's message coordinator is similar, except that it uses a terminal stone and call back handlers to invoke necessary state changes, and that received data is copied into the user's receive buffer registered through the ADIOS read interface.

3.4 Experimental Evaluation

Flexpath is evaluated experimentally using the Sith cluster hosted at Oak Ridge National Labs, and on the Windu and Jedi clusters hosted at Georgia Tech. The Sith machine is a 40 node cluster and each node is equipped with four 2.3 GHz 8 core AMD Opteron processors and 64 GB of memory. The system offers QDR Infiniband for Lustre and MPI traffic, and a 1Gb Ethernet link for communication across MPI domains.

Table 2: LAMMPS Pipeline Experimental Setup.

Data Size	LAMMPS	Helper	Bonds
76 MB	128	1	2
153 MB	256	2	4
305 MB	512	4	8
610 MB	1024	8	16

Table 3: Data sizes and core counts for weak scaling experiments

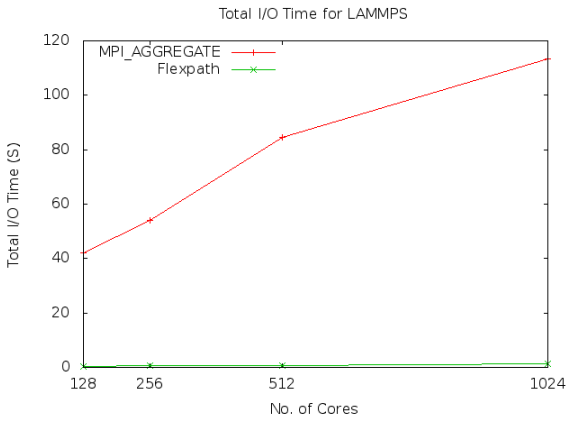
The Windu and Vogue clusters operate as separate Infiniband domains and the two clusters share a 1Gb Ethernet link for communication between the two. The nodes in both clusters contain one 2.67Ghz Intel Xeon 12 core processor and 48Gb ram.

3.4.1 Affect on Application Execution Time

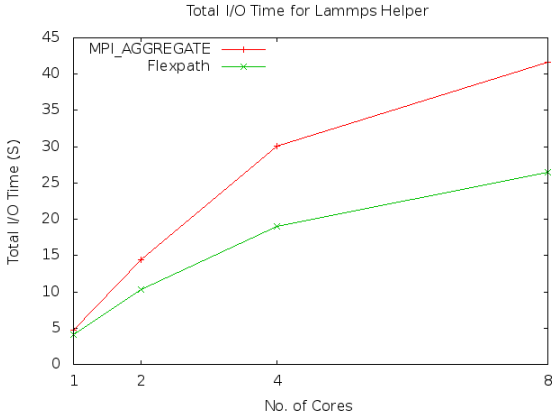
This experiment measures the effect on application level performance when constructing a workflow, using Flexpath as the data exchange mechanism. We measure time spent on output operations for each component in the pipeline and compare Flexpath's performance with that of the MPI_Aggregate synchronous disk-based method offered by the ADIOS interface. The MPI_Aggregate method is optimized for parallel Lustre I/O, and discussion on these optimizations is made available in [45].

We use weak scaling to show how the system behaves both in terms of larger numbers of participants and larger data volumes. Table 2 shows the data sizes LAMMPS produces at each output epoch, and the number of cores on which each code is executed. CSYM and CNA are serial codes that always run with an MPI size of 1, so we exclude them from the table.

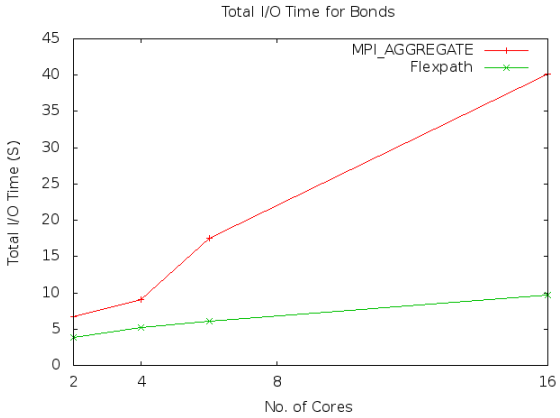
Figure 5 shows the total time each component in the LAMMPS analytics pipeline spends on performing I/O over its full execution. The LAMMPS application experiences a significant decrease in I/O time; when running on 1024 cores, it spends just over 1.3 seconds on I/O when using Flexpath vs. 117 seconds when using the MPI_Aggregate method. This is directly due to Flexpath's asynchronous nature.



(a) Lammps



(b) Lammps Helper



(c) Bonds

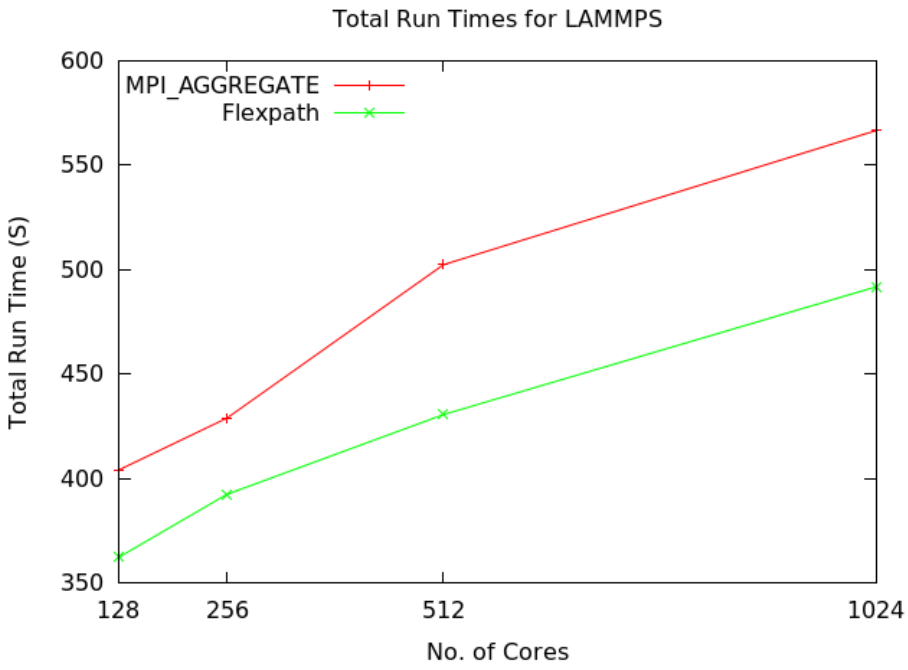
Figure 5: Total Time spent on I/O for LAMMPS and components of analytics pipeline. The asynchronous I/O offered by Flexpath drastically reduces the time an application spends on I/O operations.

Asynchronous operation also engenders reductions in I/O time for the other components in the pipeline, but the decrease is not as drastic, for several reasons: (1) the analytics components run at smaller MPI sizes than the LAMMPS application, so for each component process, Flexpath has a much larger volume of data to process and move; and (2) the Bonds and CNA components run slower than the others, so occasionally, there will be blocks in the analytics portion of the pipeline as the coordinator data stores become full. It is blocking issues in scenarios like these that motivate the notion of I/O Containers for managing analytics pipelines presented in [23].

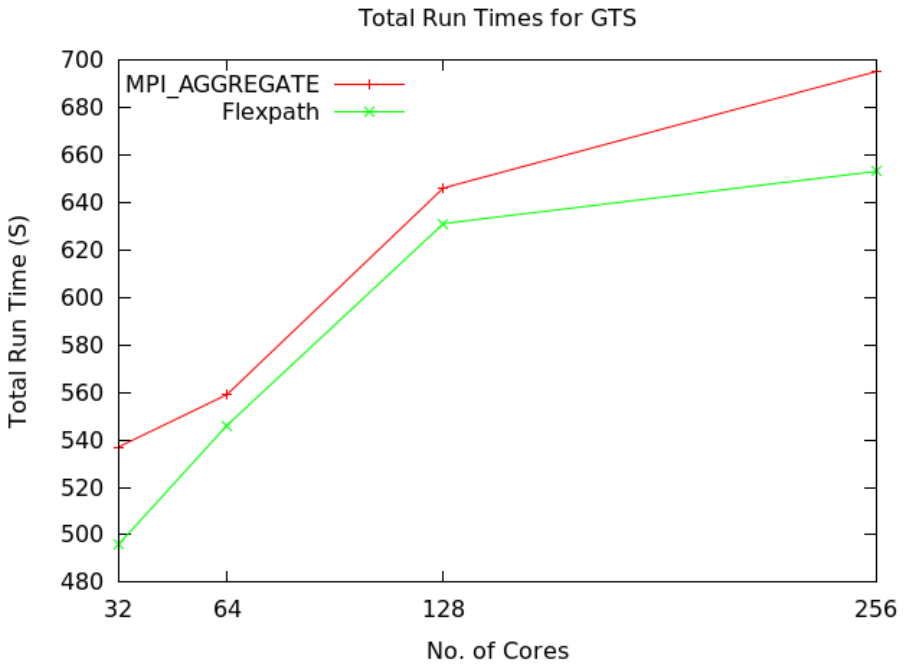
Figure 6 depicts the overall improvements in runtimes for the LAMMPS and GTS applications. The decrease in the time spent on I/O translates to decreased run times. We note that there is some disparity between the reduced time spent on I/O and the total reductions seen in run-times. This is because (i) Flexpath is an active I/O transport, so it will continue to operate and borrow CPU cycles during the application's normal execution; and (ii) with non-blocking asynchronous I/O, Flexpath data movements may collide with application level communications, e.g., MPI communications. To alleviate these effects, we can leverage the scheduling techniques described in [9], but in the experiments shown here, the decrease in I/O overhead more than compensates for these potential side-effects of asynchronous, non-blocking I/O.

3.4.2 Subscriptions and Metadata Distribution

The graph shown in Figure 7 shows the time it takes for an idle subscriber to register itself with an existing data channel. In these experiments, the CNA code sits idle and waits for an application-level control message from the CSYM code. After this message is sent, CSYM idles, and CNA activates and joins the Bonds output channel. The reason we see a linear increase in registration time here is because (i) the CNA



(a) LAMMPS



(b) GTS

Figure 6: Total Execution Times for LAMMPS and GTS applications.

code is just a serial code that must subscribe to all data from the Bonds application, and (ii) because Flexpath uses direct connections, which requires the single CNA process to establish a connection with each Bonds publisher. The time listed here also includes the time it takes for the subscribers to send their initial data fetch requests, as outlined in Section 3.3, but these costs are amortized when using the non-blocking calls the ADIOS read API offers.

The ability to use subscriptions and application level controls to perform such selective changes in data flow is an important feature when dealing with such large data volumes. This is because without such functionality, data would be delivered to subscribers before they need it and in addition, when they no longer need it. Considering the CSYM/CNA example, without this functionality, when LAMMPS is generating 610 MB of data, that would require nearly 1.2 GB of data to be transfered each epoch.

Figure 8 shows the expected costs for collecting publisher metadata and the costs expected for the subscribers in fetching this metadata from its publisher peer coordinator. This global distribution of the metadata is one feature that allows us to employ a direct-connect model without using any external metadata services. Since these costs can potentially be induced after every epoch of data, it is important to ensure that they are kept low. At 1024 publishers and 8 subscribers, we are spending less than 15 milliseconds performing these operations. To further reduce these times, it would be possible to distribute this metadata only when changes occur.

Considering that we use subscriptions to allow subscribers to receive fine-grained slices of the published data, the overhead involved with distributing this metadata is much smaller than the overhead of possibly moving large volumes of redundant or unneeded data.

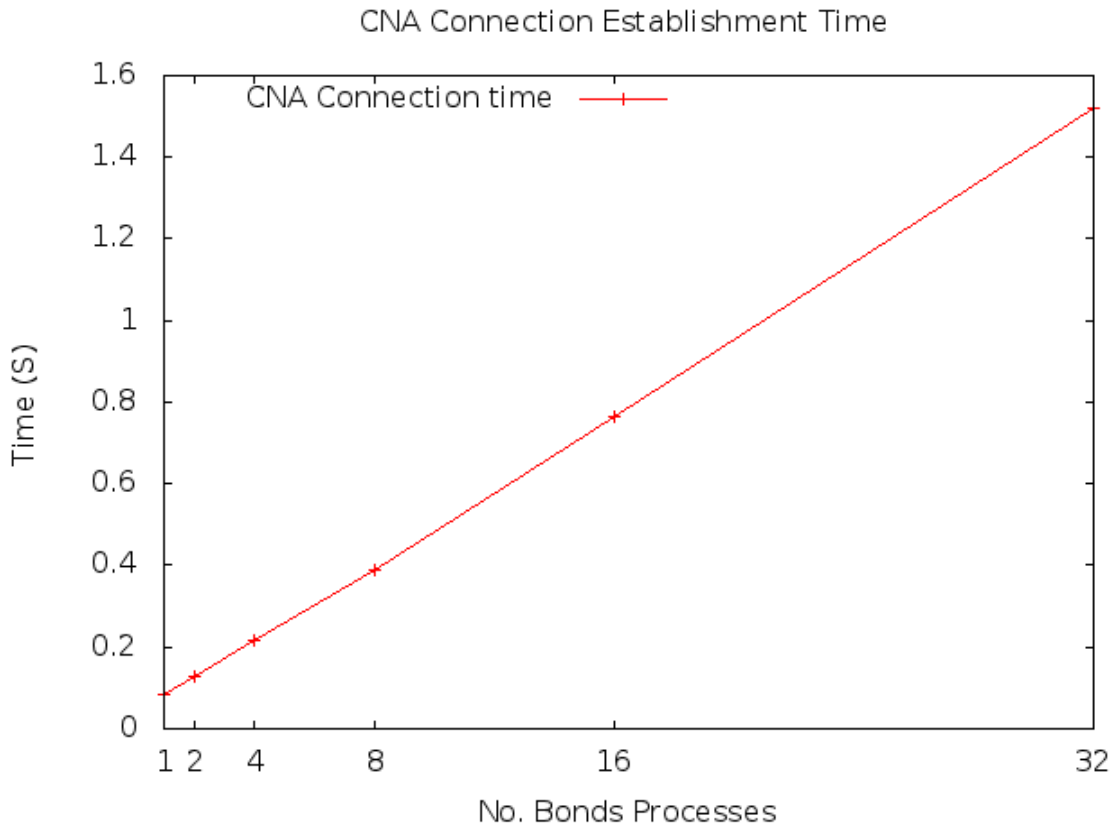


Figure 7: The time needed for 1 CNA process to join the Bonds channel.

3.4.3 Application Level Throughput

Figure 9 shows the aggregate data exchange throughput for the Flexpath system at increasing numbers of publishers and subscribers. For these experiments, we have run a two stage pipeline between LAMMPS and Lammps Helper. We conduct these experiments both on Sith and across the two Georgia Tech hosted clusters. The graphs show that in both setups, due to Flexpath's direct-connect model, we are able to achieve linear scalability as we increase the number of publishers and subscribers. This end-to-end scalability is achieved because of two key design points: (i) using subscriptions, subscribers are presented with only the slices data they request, and (ii) the use of direct connections between publishers and subscribers avoids extra data movements induced from first moving data to external brokers. Our measurements

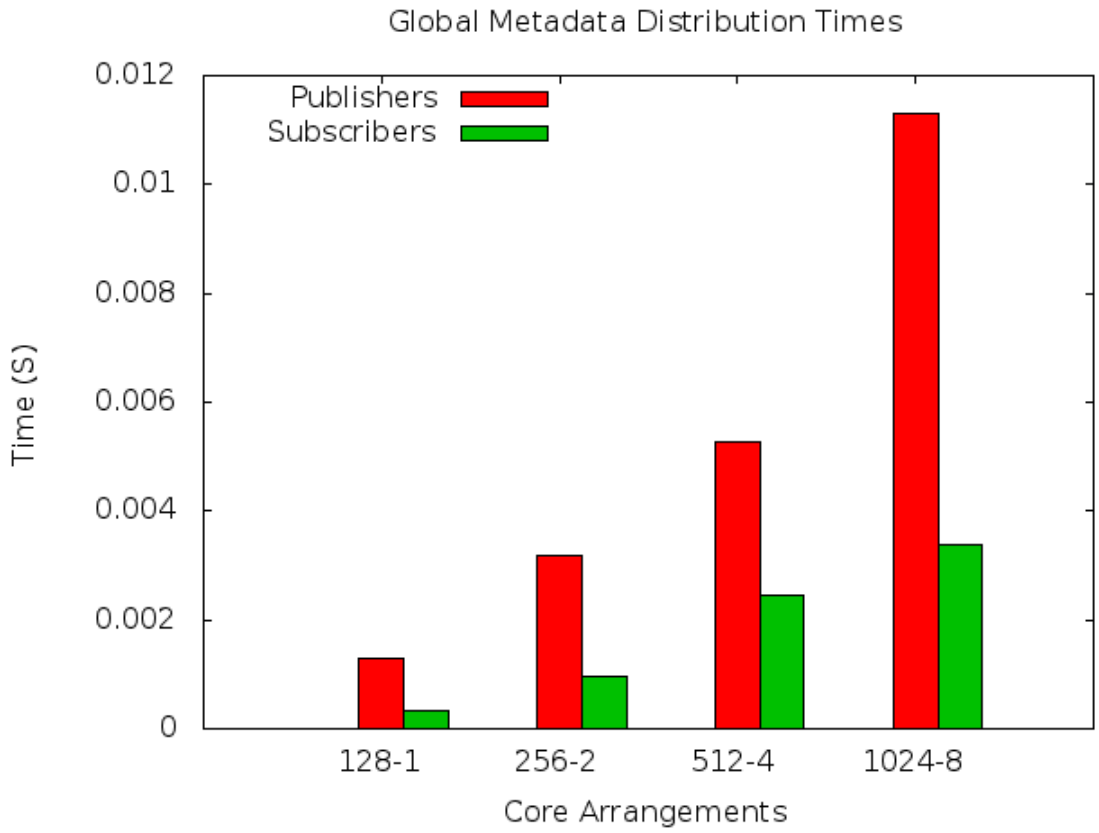
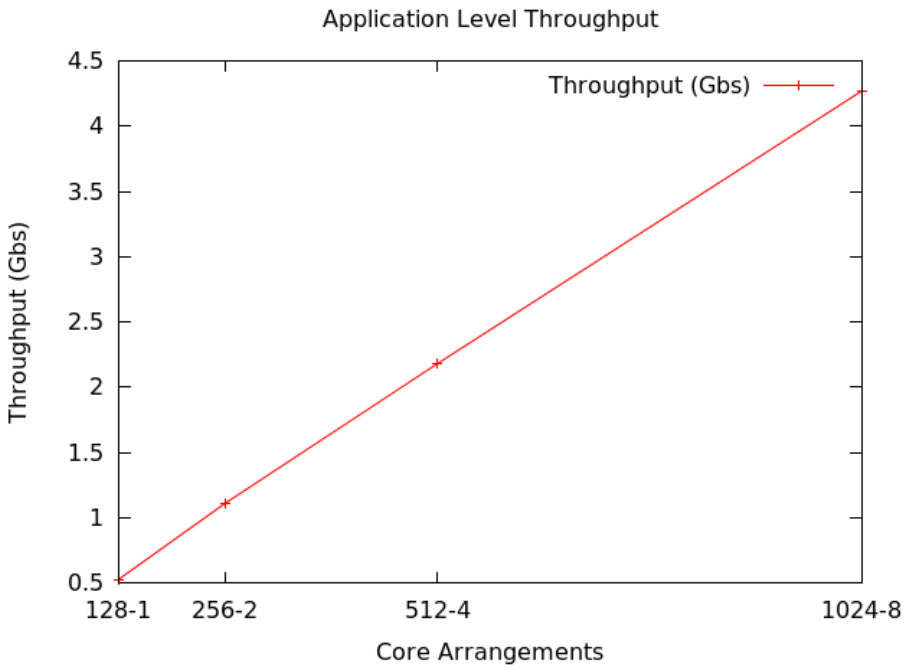
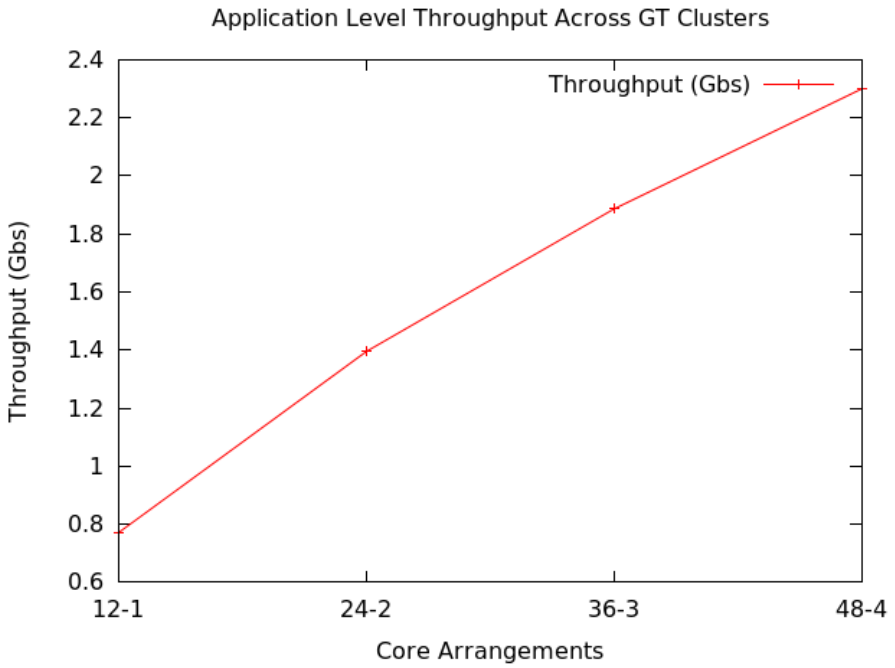


Figure 8: Time spent on collecting and distributing publisher metadata for one epoch.

for application level throughput include the round-trip times between a subscriber's fetch request to a publisher, the data transfer time, unmarshalling costs, handler invokations, and copying the data into the user provided buffers.



(a) SITH



(b) GT

Figure 9: Application Level Throughput on Sith and across Georgia Tech clusters.

CHAPTER IV

SODA: SCIENCE-DRIVEN ORCHESTRATION OF DATA ANALYTICS

4.1 SODA Overview

SODA (Fig. 10). represents the next level of abstraction for in situ workflow management, building off results of Flexpath. SODA permits developers to embed analytics tasks into a componentized, dynamically managed execution and messaging framework, called a *workstation.* Such workstations have well defined inputs and outputs [45], can be parallel (MPI or threads), and may exhibit inter-workstation dependencies. Entire I/O pipelines can be constructed by chaining workstations along their I/O paths.

SODA offers controlled resource usage, per-component orchestration, and metric-driven operation. Controlled resource usage means workstations provide and manage resources for the component mapped to it. Per-component orchestration means that a workstation can offer customized orchestration operations ensuring a component's local properties are not violated. Finally, metric-driven operation means that workstations are continually monitored to provide the runtime with the necessary information needed to enforce user or application specific metrics.

SODA also provides fault-resilient management through transactional techniques that guarantee control and orchestration actions taken by SODA do not place components into inconsistent states [46]. For example, SODA can prevent resource use until a different workstation has fully relinquished the resource. Such requirements become important as I/O pipelines scale geographically [13] as network partitions or data center outages can render parts of the pipeline inoperable.

SODA benefits code usability by allowing code developers to focus on functionality and algorithmic correctness and aleviates the need for the scientists who later use the code from the expensive tuning process and profiling runs.

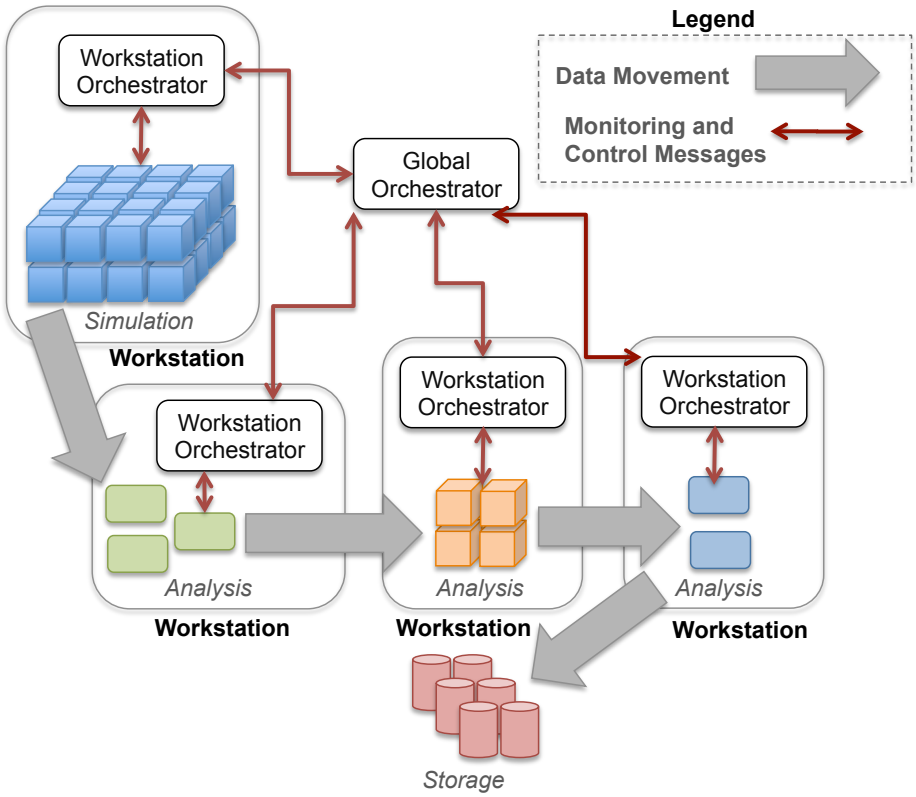


Figure 10: High-level view of the SODA framework.

SODA, with its well-defined component interfaces and programmatic orchestration API, exposes primitives for codifying SLAs by specifying appropriate actions to take when certain conditions are detected. Each workstation performs condition detection at runtime and events of interest are delivered to the orchestration hierarchy via a continuous online monitoring middleware.

Using two high end applications, the LAMMPS [53] molecular dynamics and the GTS [2] fusion simulations, along with different sets of analytics pipelines (Smart-Pointer [61] and a wave-space analysis code, respectively), we evaluate SODA with SLAs that include: (1) *bottleneck reduction* - a global performance-driven SLA that

implements "elastic workstations" to remediate detected I/O pipeline bottlenecks; (2) *data reactive* - a workstation-level data-centric policy that changes component behavior based on data feature detection; and (3) *fault recovery* - a set of policies to handle an unexpected component departure such as analysis codes on an end-user device (e.g, a laptop). Experimental evaluations show that active, SODA-based management can: (1) respond to runtime dynamics at different stack levels; (2) create and enforce SLAs at multiple granularities in an I/O pipeline; and (3) operate at large scales with low overheads.

SODA constitutes new functionality in the scientific data management domain. Current I/O staging technologies do not offer support for dynamically managing end-to-end properties of tightly coupled analytics running with high end codes. For instance, earlier data staging work runs statically profiled analysis routines in configurations sized for worst case data volumes and processing needs [66]. Similarly, our recent supercomputer simluation "in-situ" analytics work [67] schedules and manages only the analytics actions taking place on individual compute nodes without concern for the I/O pipeline end-to-end properties affected by such nodes.

SODA, building off of Flexpath, is designed for 'in transit" workflows and is mostly concerned with coarse-grained orchestration operations where end-user goals can be met by re-allocating staging resources. However, the SODA framework provides a hierarchical management model and implementation which serves as a solid basis for CoApp's goals.

4.2 SODA Framework

SODA is a set of run-time abstractions for dynamically orchestrating science applications and their associated analytics executables. Analytics executables are encapsulated in *workstations* that are connected along their I/O paths to form an *I/O pipeline*. Orchestration is conducted through an orchestration hierarchy and is guided by a

flexible event-driven monitoring and control infrastructure. Fig. 10 depicts SODA's conceptual model.

4.2.1 Assumptions and Desired Properties

Given the assumptions outlined in Chapter 1 and the set of challenges and application characteristics outlined above, SODA based pipeline orchestration must meet four design goals. Given the large variety analytics code characteristics and the dynamics they experience at runtime, it is impractical for a single entity to understand all analytics in some composed I/O pipelines. Therefore: *(1) orchestration routines and policies should be customizable on a per-workstation basis.*

To make decisions at run-time, orchestration functions require information about when and what actions should be performed. Gathering this data requires continuously monitoring pipeline components for their progress, behavior, and the physical resources they use. Using this information, orchestration actions can be invoked in a timely manner. Therefore: *(2) orchestration is guided by user-determined metrics driving per-workstation and cross-workstation (i.e., global) orchestration policies.*

Ideally, analytics pipeline components should be decoupled along the time and space dimensions allowing correct operation depending only on necessary data availability (i.e., from disk or via the network). With well-defined input and output interfaces, analytics actions can be allowed to run independently as separate applications (i.e., components), and enter and leave the pipeline as needed. This enables using entirely different, dynamically swappable analytics codes without requiring them to be integrated into a single executable. Therefore: *(3) analytics codes should operate in a componentized fashion.*

Orchestration on one component can jeopardize the execution of other components. For instance, consider trading resources between two analytics components

when recovering from some detected bottleneck. A failure can occur if one component, using incorrect resource state data, attempts to use a resource that has not been fully relinquished by another component. Therefore: *(4) orchestration operations must be reliable and be resilient to failure.*

By meeting these design goals, SODA can be used to realize (1) customized per-component and global management policies; (2) enabled by online monitoring of the varied metrics of relevance to different policies; (3) componentized operation consisting of swappable codes; and (4) made resilient to failure via transactional control methods.

4.2.2 Conceptual Model

4.2.2.1 Workstations

A workstation, depicted in Fig. 11 allows analytics tasks to be embedded into a dynamically managed messaging and execution framework. Indeed, it could be considered as a high-performance implementation of an application service within a Linux Container. The workstation's I/O interfaces are similar in concept to those used in modern Service Oriented Architectures (SOA). The workstation is comprised of a set of *active replicas* and a *workstation orchestrator* overseeing its execution.

Active Replicas. Unlike the replication techniques used in fault tolerant systems where replicas have identical internal states[31], active replicas in workstations are key to obtaining scalability: with traditional replication, each replica performs redundant computations on the same data items whereas active replicas perform their computations on different epochs of data assigned to them. For the use-case discussed in Section 4.4.1, data is assigned to active replicas in a round-robin fashion, but additional communication patterns can be supported. Using active replicas, a workstation orchestrator can increase its degree of parallelism by spawning a new replica. While this is similar to how Map-Reduce jobs scale, note that an individual data epoch may only be able to be processed by a fixed process count and that scalability comes from

overlapping processing of different epochs.

Workstation Orchestrator. The workstation orchestrator provides several functions. First, it collects and organizes relevant monitoring data from its active replicas and delivers this information to a higher-level orchestrator. Second, it provides metadata services for its replicas and contains end-point information for replicas in neighboring workstations. Third, it contains potentially custom management primitive implementations, described next, which allow them to respond to management requests from higher-level orchestrators.

4.2.2.2 Orchestration Constructs

Hierarchical orchestration affords three primary benefits. First, such hierarchies can be scaled with ease [59]. Second, distinct per-workstation orchestrators can offer customized management routines and separate their local, per-component management states from global state about entire I/O pipelines. Third, the hierarchy helps define authority. A global orchestrator is responsible for operations that re-organize entire pipelines. Workstation orchestrators are responsible for operations affecting only their components and resources and respond to management invocations from higher-level (global) orchestrators.

The following core management primitives enable constructing higher-level policies and operations:

- **Increase Workstation:** allocate more resources to a workstation with the goal of increasing scalability.
- **Decrease Workstation:** deallocate resources to a workstation that may be relatively over-provisioned.
- **Offline Workstation:** remove all resources from a workstation and redirect dataflow from upstream to disk because it is no longer feasible to run a workstation online due to network partition or insufficient resources.

While the per-workstation orchestrator actions listed above are invoked by a global orchestrator, the concrete steps needed to execute these actions within a workstation can be customized on a per-workstation basis. For example, when told to "increase", a code that cannot operate on data epochs out of order could "increase" by killing its existing active replicas and spawning with a greater process count.

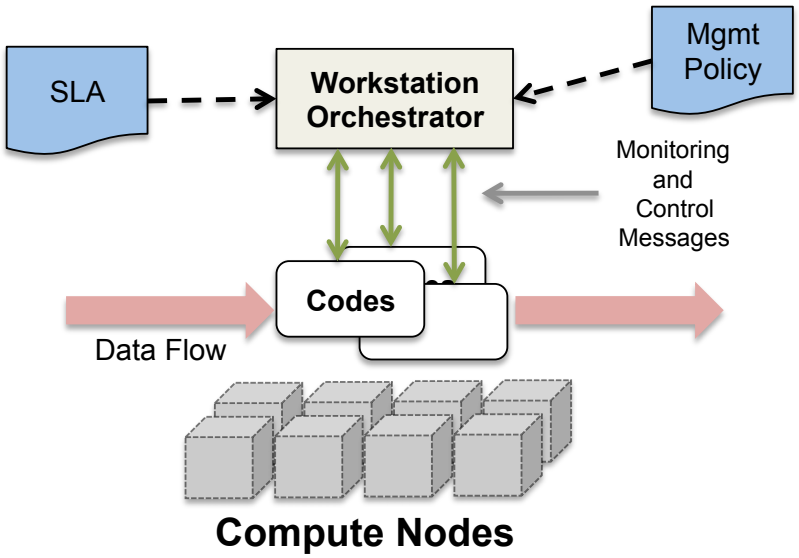


Figure 11: Workstation abstraction.

4.3 Implementation

4.3.1 Workstation

4.3.1.1 Active Replicas

The implementation of SODA workstations leverages the widely used ADIOS read and write interfaces [45]. Using these interfaces, analytics codes can specify their data requirements and establish communication via a virtual file name serving as a named communication channel. To accommodate orchestration at runtime, the Flexpath [22] ADIOS transport, which allows for online analytics routines to exchange data, has been extended to accept and process management messages and state-change notifications from the replicas' designated orchestrator. We also modify the

ADIOS interface to expose a communicator analytics applications can use to interact directly with the workstation orchestrator.

Flexpath publishers (ADIOS writers) maintain a queue for each neighboring Flexpath reader replica in a downstream workstation to hold data epochs. Writers then assign data to these queues in a fashion determined by the reader workstation. The current implementation supports round-robin assignment including the case where one replica consumes all of the work for an existing replica. This is explained in more detail in Chapter 4.4. Orchestration operations can also lead to internal load balancing actions to offload work from overly filled queues. Conversely, with a "decrease" operation, it can re-assign existing work to the remaining replicas.

4.3.1.2 Orchestrators

Orchestrators are written to be run as stand-alone executables. Users can create custom orchestrators and specify SLAs using a programmatic API described in Section 4.3.2. When global orchestrators detect conditions of interest, they invoke commands on workstation orchestrators and then distribute any important state changes to subsequent workstation orchestrators that require knowledge of such state changes. Workstation orchestrators are responsible for implementing the commands invoked on them by global orchestrators and for performing internal actions on the resources and replicas they manage.

4.3.2 Orchestration Interface

The basic primitives listed in Section 4.2.2.2 are exposed as a C interface. Developers use this interface to create custom orchestrators if needed. SODA ships with some default implementations to automate elasiticity and recovery from a failed replica. To meet an SLA at a global orchestrator, the orchestrator can receive monitoring information as events and carry out chained primitives to perform actions like resource trading. When invoked, an orchestration primitive triggers a set of transactional

protocols that indicate a participant's progress and distribute any state changes.

In our current implementation on the Titan machine at Oak Ridge National Labs, launching replicas is conducted as follows: the workstation orchestrator constructs an *aprun* command as a text string writing this command to a file. The *PBS* job script (a feature of the PBS job scheduler), regularly scans each workstation orchestrator's file for commands. When one is present, it reads and executes it. This implementation is due to the constraint that only the root node of the job, which executes the PBS script, can launch MPI-enabled applications on the compute nodes. While this illustration and the use case presented in Sec. 4.4.1 focus on output queue build up, management could also be triggered by other factors such as memory consumption or CPU utilization.

4.3.3 SODA Information Bus

Monitoring, control, and state change messages are delivered via the *SODA Information Bus*, or SIB, implemented using the EVPath [28] event-driven messaging library.

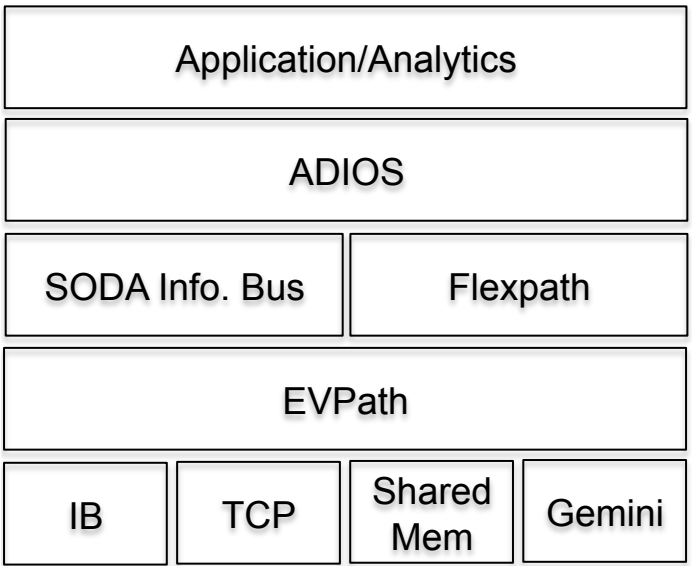


Figure 12: SODA software architecture.

Orchestrators and replicas are connected via the SIB's overlay graph where workstation orchestrators serve two roles: (1) aggregation points for monitoring information, execution metadata, and runtime state information before delivery to the global orchestrator; and (2) as orchestration operation entry points into a workstation and the delivery point for state change notifications (i.e., state that determines from who replicas read data) from neighboring workstations.

Global orchestrators serve as the root of the SIB and accept and organize messages from all other orchestrators. To ensure strong runtime state information consistency, the current implementation passes all messages relating to state changes through the root.

In the case of parallel replicas (i.e., MPI based analytics codes), rank 0 is designated as the message recipient from the workstation orchestrator. It then uses MPI to disperse the messages to the remaining ranks. This takes advantage of MPI's optimizations and reduces the number of connections a workstation orchestrator has to maintain.

4.3.4 Fault Detection and Recovery

The current implementation detects faults in two ways. The first uses application-level progress indicators delivered via periodic heartbeat messages from an application replica to its workstation-level orchestrator. The second allows the orchestrator to receive a notification from the kernel when the connection between an orchestrator and a replica has been broken. Method 1 does not rely on a specific messaging technology (e.g., sockets) and can work for a variety of underlying network interconnects with the disadvantage that the orchestrator must propagate failure notifications through the SIB to interested parties. Method 2 allows for any component interacting with it (orchestrators or other replicas in the pipeline) to receive the notification without waiting for failure alerts to propagate through the SIB. Both methods are explored

in our current investigation because they are familiar to end-users and have well-understood characteristics. Future work will explore more robust fault detection [52, 17] and diagnostic [59] mechanisms.

The specifics of how to recover from a component fault is left to the user via API calls in the associated orchestrator. For example, issuing an "offline_workstation" operation or spawning a new replica on spare resources (an "increase_workstation" operation). The SODA framework does provide some fixed options configured at registration time specifying whether components can deal with data loss. For a visualization component operating in a "streaming" fashion, it might be able to tolerate a few missed frames. For these, we can redirect the data to other replicas that have not failed or discard the data if none are available. For codes where missing output epochs could render scientific results invalid, such as stateful codes, we allow for upstream data publishers to buffer the data, either in memory or by leveraging on-node storage (SSDs) via EVPath "storage stone" facilities, until the failed replica has recovered.

4.4 Experimental Evaluation

Experimental evaluations are conducted using two machines: (i) the Titan supercomputer hosted at Oak Ridge National Labs and (ii) the Maquis cluster hosted at Georgia Tech. Titan consists of 18,688 compute nodes each containing 16 cores and 32Gb memory for a total of 299,008 cores and a peek performance of over 20 petaflops. The Maquis cluster is a 16 node Infiniband cluster with each node having two Intel Xeon quad core processors and 8GB of RAM.

The LAMMPS molecular dynamics simulation and the SmartPointer analysis toolkit serve as our application drivers for Titan. We construct two policies to demonstrate the benefit of the SODA approach and to assess the active management overheads. We run the GTS fusion simulation on Maquis and execute the spectral analysis

(FFT) code on a machine at a remote location thereby allowing us to test the system's behavior when the pipeline is geographically distributed. We cannot conduct geographic experiments on Titan as its security policies and firewall settings prevent this.

4.4.1 Application Drivers

4.4.1.1 LAMMPS and SmartPointer

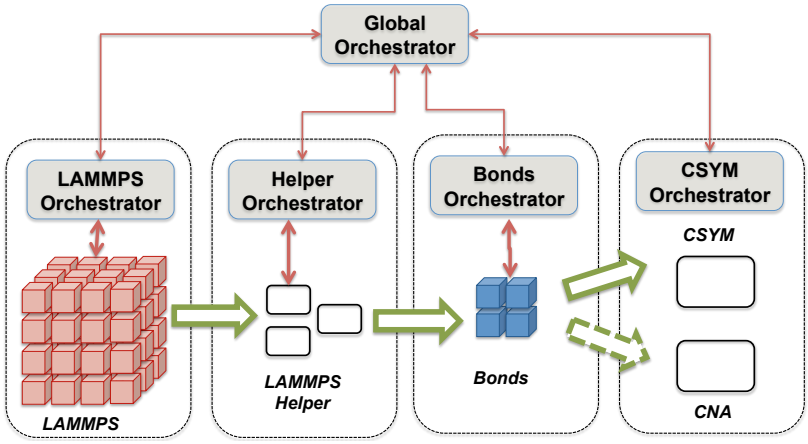


Figure 13: I/O Pipeline for LAMMPS with SODA

Figure 13 depicts the I/O pipeline constructed for the LAMMPS (Large Scale Atomic/Molecular Massively Parallel Simulator) [53] science application using the SmartPointer analysis and visualization toolkit, both of which were discussed in Chapters 2.1.1 and 2.1.1.1 respectively.

4.4.1.2 GTS and FFT Analysis Code

As an alternative application example, to demonstrate the more general utility of SODA, we also evaluate our framework with GTS [2], a plasma fusion simulation with an implementation that exploits coarse grained process level parallelism using MPI and more fine-grained thread-level parallelism using OpenMP. This "particle in cell" code has different output frequencies for both particles and mesh-level statistics. To examine the dynamics involved, in particular dangerous transient effects that

might damage a real reactor vessel, it is useful to dynamically evaluate and characterize particular trends on the inner and outer plasma edges. Unlike the LAMMPS case, these transients are not as algorithmically identifiable. Secondary analysis methods are used to infer their existence and then much more detailed inspection involving direct interaction with the physicists is used to further the investigation. The GTS analytics pipeline is a spectral code based on the AMD Core Math Libraries implementation of FFT that ingests the phi and Z-ion output arrays from the simulation.

4.4.2 Management Policies

For LAMMPS and its SmartPointer pipeline, we have constructed two policies:

Quality of Service (Global): the Bonds and CNA codes are slow components compared to the LAMMPS simulation with CNA being the most expensive. Bonds executes on every output epoch whereas CNA executes only when CSYM reports a crack. Depending on the output frequency or how soon a crack is detected, these codes can become bottlenecks in the pipeline. We create a policy that monitors queue lengths such that if the global orchestrator detects a growing queue length reaching a size threshold on some output workstation, we perform an "increase" operation spawning additional replicas for the slow component. In this pipeline, it is either Bonds or CNA. This represents a global policy seeking to balance pipeline components to ensure healthy end-to-end throughput. It also allows for the pipeline to run without needing to carefully provision both Bonds and CNA codes; the system can handle the provisioning when needed. While this illustration uses queue lengths, orchestration could also be triggered by other factors such as memory consumption or CPU utilization.

Data-centric (Local): requires application introspection into the data based on the CSYM and CNA components. In contrast to the first policy, the analysis functions report the metric of interest (CSYM detects a crack) and the orchestration actions

(kill CSYM and run CNA) are triggered by the workstation-level orchestrator. This policy ensures data quality via correct execution of pipeline analysis functions.

For the GTS and FFT example, the analysis running on an end user's machine is connected with the simulation code over a wide area network. We evaluate workstation output latency when faced with an unexpected component departure, e.g., when an end user terminates analysis. Three recovery polices are tested. Each involves failure detection on the remote machine and spawning a recovery replica on the cluster running the simulation. If components need a data guarantee, they can pay the costs for it. Less critical codes can avoid these extra costs by tolerating missing output epochs. The first policy allows for data loss while the second avoids it. In these two cases, the recovery replica is launched in response to a failure notification. The third policy takes advantage of over-provisioning by the workstation spawning an additional FFT replica on the compute cluster that remains idle until its orchestrator detects a failure.

4.4.3 Quality of Data Policy and Microbenchmarks

SODA-orchestrated I/O is beneficial, but it also imposes additional overheads on I/O pipelines. The following measurements assess the protocol overheads and compare costs at different scales for operations invoked at different orchestration hierarchy levels. The measurements shown elide the base constant cost of process instantiation (e.g., for a workstation increase), as that cost is specific to the underlying machine's job scheduler rather than the implementation and protocols specific to SODA. On the Titan machine, we have seen highly variable launch times, sometimes higher than 30 seconds.

Orchestration costs are governed both by the inherent properties of the management methods chosen and their underlying protocols and by the scales of interacting workstations. The latter is due in part to the "direct connect" nature of

Helper Size	2x16	4x32	8x64
Orchestrators	0.12s	0.126s	0.111s
Helper	0.039s	0.089s	0.158s
Csym/CNA	0.024s	0.031s	0.027s

Table 4: Increase Command Protocol Overhead

Bonds Size	1x256	2x256	3x256
Orchestrators	0.051s	0.074s	0.063s
Bonds	0.026s	0.05s	0.072s

Table 5: Data-Centric Command Protocol Overhead

the Flexpath transport used in the implementation of SODA: Flexpath obtains high cross-workstation throughput by directly connecting the parallel entities of a previous workstation to the parallel entities of a subsequent one. This also means, however, that the cost of distributing certain state changes (e.g., workstation increase) is affected by the size of the neighboring workstations as each of their parallel entities must be notified about this state change.

Table 4 shows the modest protocol overheads for an *increase* operation on the Bonds workstation. The row titled "Helper" represents the time it takes for the Helper workstation to distribute the Bonds state change. This includes the time it takes for the workstation orchestrator to send the state change to each replica (rank 0), and the time it takes for rank 0 to broadcast this change to the other ranks. The row titled "Orchestrators" is the total time spent for all messages between the global and workstation orchestrators to trigger the operation, and to distribute the state changes. As expected, use of an orchestration hierarchy allows for good scalability, demonstrated by the fact that for measurement, we are increasing the number of Lammps Helper processes by a factor of 4, but only see a growth of $2x$ in terms of protocol cost. Since these management actions do not affect the number of orchestrators, the communication between global and workstation-level is not affected by scale.

Table 5 shows the cost of the protocol used to enforce the workstation-level data-centric policy, i.e., switch off CSYM and activate CNA. This represents a control loop triggered by the workstation orchestrator (when CSYM detects a crack in the modeled material) that results in a change in the data flow (Helper redirects its output data to the CNA component). We see scalability traits similar to that of the increase operation; the reason this command takes much less time to execute is because CNA is a single replica serial component, so the size of the state message is much smaller.

4.4.4 Throughput Measurements: QoS Policy

This set of measurements demonstrates the utility of a representative performance-based management policy. We compare the throughput of the SODA-orchestrated I/O pipeline against that of an unmanaged pipeline, where throughput is represented as a time series in 30 second increments along the x axis, and the y axis represents the count of output epochs emitted by the code during that 30 second interval.

Fig. 14 shows the baseline, unmanaged execution, for a LAMMPS simulation running on 8192 cores and a pipeline comprised of 64 Lammps Helper cores, 256 Bonds cores, and 1 CSYM core. The graph shows that as the output queue for Lammps Helper fills up, LAMMPS' throughput drops significantly. This is because it has to block on its output actions that must wait on queue space to free up. LAMMPS' throughput converges to that of Bonds, the slow component, effectively dropping end-to-end throughput to a third of the ideal target.

Fig. 15 depicts the throughput improvements for a set of QoS-orchestrated runs that demonstrate the SODA runtime's ability to provide elasticity at scale. Experiments are run at three scales, with the process counts displayed in Table 6. For each experiment, the slow workstation is detected and increased by a replica with the number of processes equal to the size of the initial replica. For these runs, the crack in the material did not materialize until the end of the run, so that the main

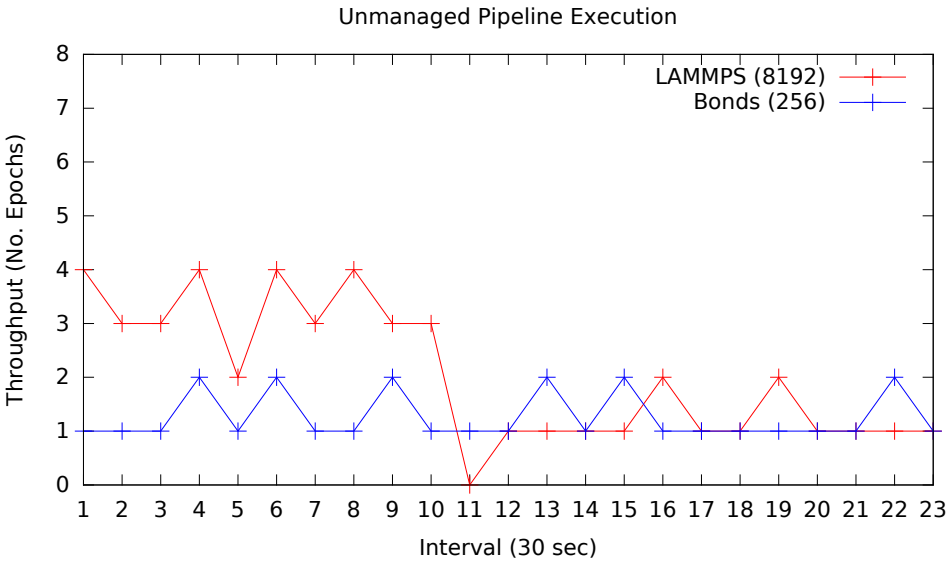


Figure 14: Throughput degradation for unmanaged pipeline.

	LAMMPS	Helper	Bonds	CSYM
Fig 15(a)	8192	64	256 to 768	1
Fig 15(b)	4096	32	128 to 384	1
Fig 15(c)	2048	16	64 to 192	1

Table 6: Core Counts for Throughput Experiments

component needing an increase was the Bonds code. Fig. 15(a) shows the throughput improvements for running with 8192 Lammps cores. The vertical lines represent when Bonds is increased. For this run, we see that after the first increase (two Bonds replicas total), we see an improvement in Bonds throughput. However, an additional increase is needed for Bonds to match the throughput of the LAMMPS simulation. After this second increase (3 Bonds replicas, 768 cores total), we see that Bonds can achieve a higher throughput than the LAMMPS application, as it now has sufficient resources to start to drain the data that has built up in the queue.

Figure 15(b) shows a similar result, where after three increases, Bonds maintains a slightly higher throughput than the LAMMPS simulation. However, speedup is insufficient to fully drain the queue in Lammps Helper, so the Bonds code executes

somewhat longer. We see a similar phenomenon in Fig. 15(c), where the global orchestrator does not increase the Bonds workstation by an additional replica because the stated policy is to trigger an increase only when two conditions are met: (1) a maximum queue length of 10 in one of the Helper output queues, and (2) a growing maximum queue length for 3 consecutive measurements. For the latter two runs, condition (2) did not trigger. This example illustrates the utility of explicit policy specification. An alternative policy omitting the second condition would have triggered the additional Bonds increase. An energy-conscious policy might prefer a slight extension in execution time over the additional energy consumed by using additional nodes.

Fig. 16 displays the changing queue length, the metric on which we base throughput management, for an experiment with the same setup as in Fig. 15(a). This represents the maximum queue length in the Lammps Helper workstation's output queue for the Bonds workstation. Here, the x axis represents the output epoch, and the y axis represents the max queue count when that output epoch is inserted into a queue. As is evident, the stated management policy is having the desired effect on its metric of interest.

4.4.5 Fault Recovery Policy

The experimental results reported next have two purposes. First, we want to understand how SODA's fault recovery operations for an unexpected component departure affect the applications relying on them. To quantify this, we look at workstation latency, which measures the time it takes for a workstation to emit an epoch of data. Second, we want to demonstrate the flexibility the SODA constructs offer to developers for choosing which tradeoffs make sense for their executions. For all three cases, we use a heartbeat to detect a component's departure, where heartbeats are configured to run in 10 second intervals, and a component is considered failed after missing

three consecutive heartbeats.

Fig. 17 displays the changes in workstation latency for three different fault-recovery mechanisms. The x-axis represents the epoch number for a workstation, and the y-axis represents the length of time between a step and the previous step. The first time step for each has a high latency, since we use the application start time as the base.

The first graph, Fig. 17(a), shows the workstation latency when recovering from a fault, but allowing for data loss, which is represented by the discontinuity for the FFT line. This has the lowest latency across all three because the previous (in other words, the older) time steps are simply dropped. Allowing for dropped epochs of data becomes more even more beneficial with configurations where it is infeasible, in terms of memory requirements, to buffer multiple timesteps of data.

The second and third graphs show the changes in latency when avoiding data loss. As expected, we see a higher latency than when allowing for data loss as the older timesteps stay in the queue. The third graph has a lower latency during the failure and recovery phases, because the over-provisioning of the codes allowed the FFT replicas to register with the the orchestrators and get the necessary metadata to join the stream at the start of the pipeline execution. This process accounts for the roughly 6 seconds difference between the third and fourth graphs.

In all three measurements, the dominating factors concerning latency are the heartbeat intervals, the number of missed heartbeats used to detect a failure, and the GTS application's own I/O cycle. For the latter, this is a result of the Flexpath publisher component checking for notifications from the workstation orchestrator when calls are made into the ADIOS interface. As the graph shows for the GTS latency, I/O epochs occur about every 8 seconds. Lower latency could be obtained by using shorter heartbeat intervals.

4.4.6 Discussion

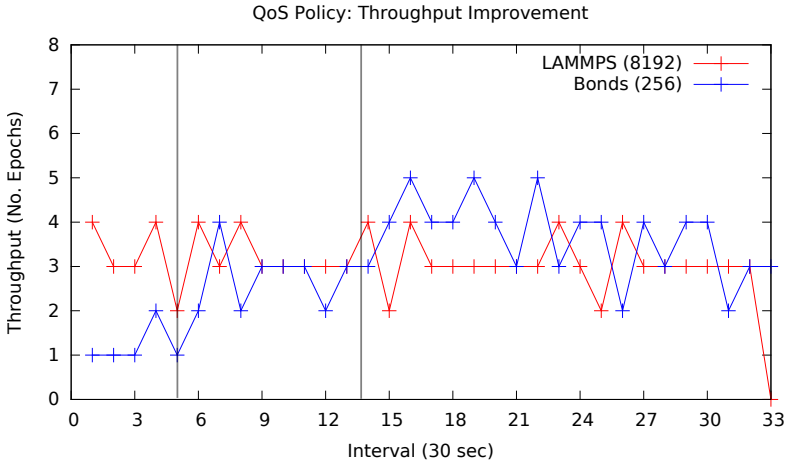
SODA-orchestrated I/O pipelines provide elasticity at scale, data-centric management opportunities, and configurable fault recovery options for the online analytics pipelines constructed for high end simulations. Through active replication, *elastic workstations* can automatically adjust their data processing throughput to match application output rates and the behavior of other workstations with which they have been composed. Performance-driven policies like those pertaining to throughput can be replaced with alternative policies concerned with end-to-end latency, caps on energy use, or others, without affecting the implementations of individual analysis components. By exposing SODA controls to applications, orchestrators' actions can be based on the receipt of application-specific events, thus enabling a variety of application-specific SLAs and management policies. By taking advantage of a decoupled pub/sub data movement substrate with internal buffering capabilities, we can provide flexible recovery options to applications so they can handle faults like unexpected replica departures.

The performance results shown above demonstrate the superiority of managed vs. unmanaged I/O, guided by simple policies realized with low cost management structures. While able to scale to the high end machines currently available to our research, the current management policies implemented for SODA assume each workstation running on its own dedicated resources, separate from those used by the application. Management actions that involve scheduling or resource sharing [67] remain part of our future work.

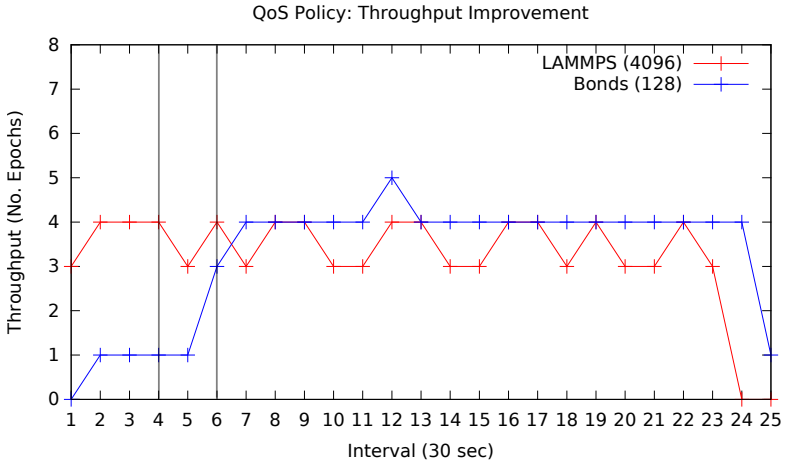
4.5 Conclusions

The SODA framework presented in this chapter permits users to embed their scientific data analytics tasks into a dynamically managed execution environment that (1) continually monitors analytics components for metrics of interest, (2) allows users to

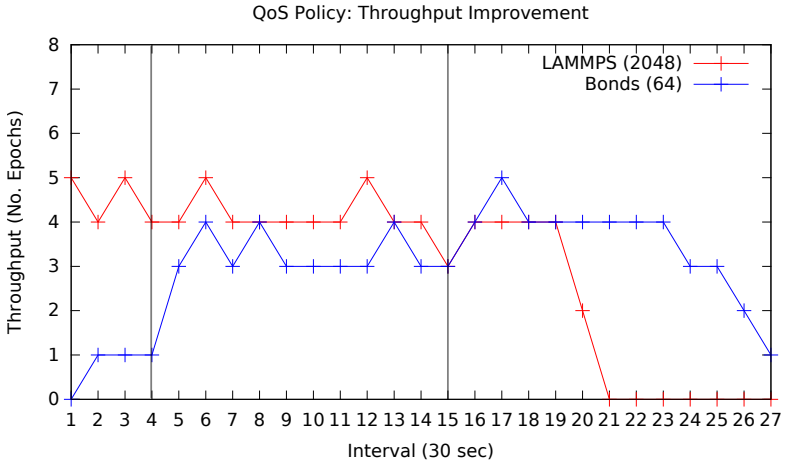
specify management policies and enforcement operations at different granularities of the pipeline, (3) provides elasticity at scale for their analytics tasks, and (4) does so efficiently with low management overheads. The utility of SODA is demonstrated with three policies associated with I/O pipelines consisting of realistic science applications and analytics pipelines: (1) a global "quality of service" policy permits an I/O pipeline to recover from a poor initial resource allocation; (2) a "quality of data" policy operating at workstation-level allows for new analytics tasks to be injected into the pipeline to respond to the richness of features discovered in the data; and (3) fault recovery policies handle an unexpected component departure in a geographically distributed pipeline.



(a) 8192 LAMMPS cores with 1 to 3 Bonds replicas of size 256



(b) 4096 LAMMPS cores with 1 to 3 Bonds replicas of size 128



(c) 2048 LAMMPS cores with 1 to 4 Bonds replicas of size 64

Figure 15: QoS Policy: throughput improvements.

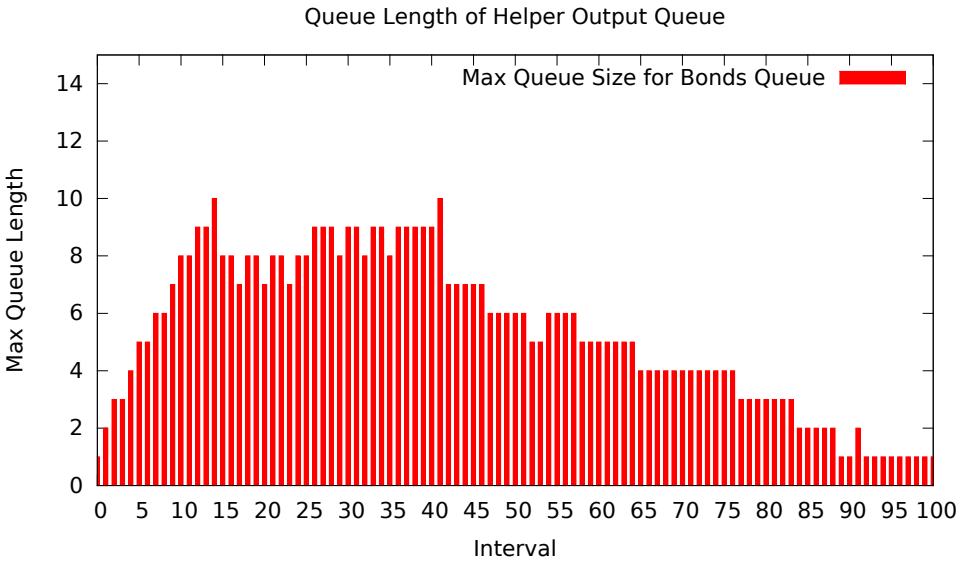
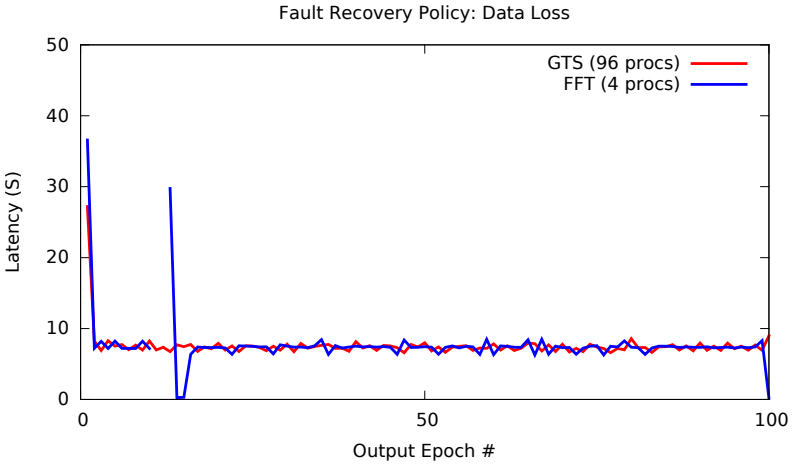
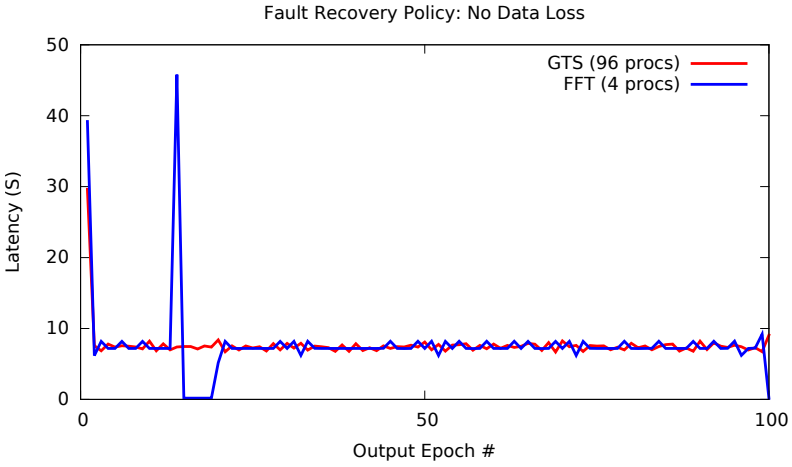


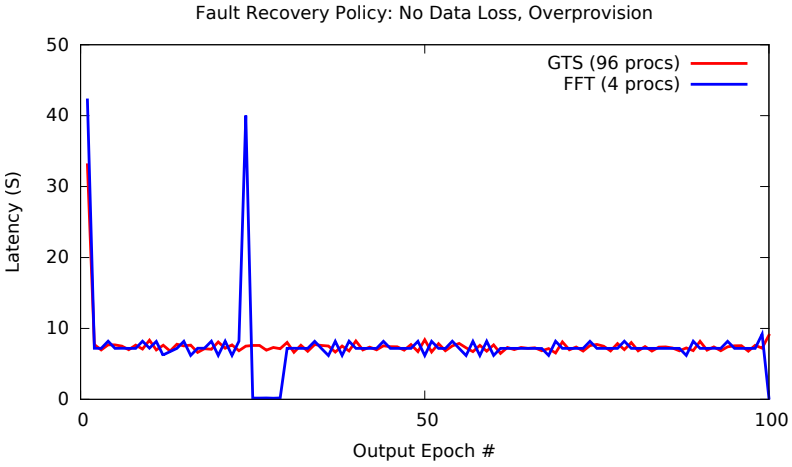
Figure 16: Change in max queue length for Helper Workstation.



(a) Fault Policy: Data Loss



(b) Fault Policy: No Data Loss



(c) Fault Policy: Overprovisioning

Figure 17: Failure recovery policy affect on latency

CHAPTER V

COAPPS: MIDDLEWARE FOR IN SITU ANALYTICS

5.1 CoApps Overview

CoApps is a demonstration of a full-featured, orchestrated, in situ workflow. By using CoApps, users can run their workflow components in all styles of in situ without needing to change their code to use different middleware or run times targeted for specific in situ definitions. Building off the dynamic nature of the previous work, CoApps can use position independence to perform resource sharing and node consolidation to better meet user driven goals.

CoApps derives from, and shares concepts with, the more familiar programming construct of co-routines. Like co-routines, CoApps are capable of holding state across invocations and they enable fine-grained control over their execution, meaning you can yield and resume them as needed without corrupting their internal state or correctness of results. CoApps are also capable of invoking other CoApps (like recursive workflow models) and returning results to the parent process. These features do not preclude CoApps from sharing the address space with the parent process and operating as more traditional co-routines, or even as standard in-line functions.

In contrast to co-routines, the CoApps abstraction is designed to let analysis components operate in a completely separate process space from the parent application allowing for concurrency. By giving analysis components some degree of independence, we can also manage them separately from the parent application. For example, we can independently scale them or even implement them to use hardware or run times the core simulation is not equipped to use, such as using the GPU in a CPU-optimized environment or visa-versa.

At an abstract view, CoApps have some resemblance to the functionality Linux Containers (LXC) [30, 4], and their native management frameworks like Docker[11], offer. One critical distinction is that we seek to do everything in user space instead of relying on virtualization technologies. There are several important reasons we do this. First, such virtualization technology is not readily available on a large number of HPC leadership machines, like Titan at Oak Ridge National lab. Second, by doing it all in user space, we can give applications and end users direct control over their resources to give them great flexibility in how they manage them. A primary goal of virtualization technologies is to provide a guarantee of isolation as physical resources are shared across users, while the goal of CoApps is to provide for position independence and collocation of analysis workflow components. In fact, there is nothing preventing CoApps from being embedded and deployed using virtualization technologies; CoApps would provide the application connectivity and SLA management and virtualization technologies could enable cross-user node sharing and isolation properties.

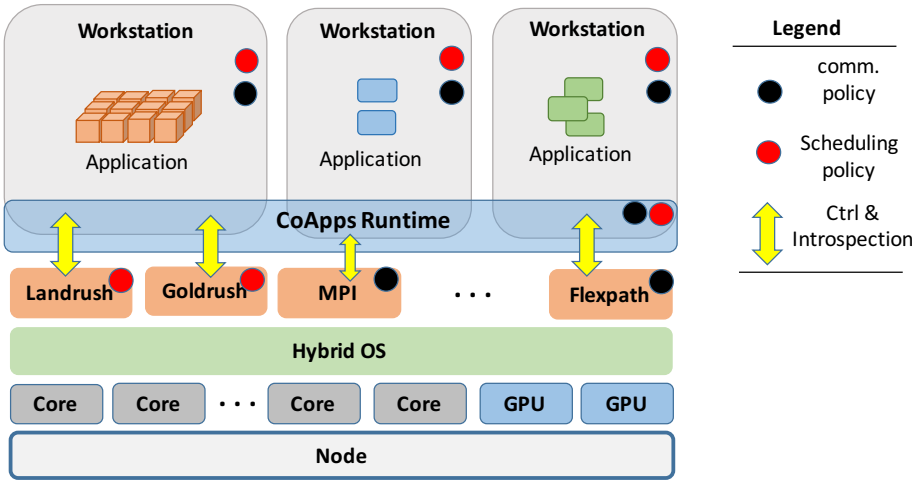


Figure 18: High-level view of the CoApps Run Time.

Figure 18 depicts the conceptual view of CoApps implemented within the SODA orchestration framework. CoApps generalizes the active replica concept from SODA by taking into consideration collocation opportunities when making orchestration

decisions. To enable this, we had to make several modifications to the initial SODA model, both in the assumptions made and the software artifacts.

5.2 CoApps Design and Implementation

A key assumption made in the initial SODA model was that nodes were relatively small and homogeneous, and resources would be allocated at whole node granularities. As we move towards exascale, researchers are testing platforms with a fewer number of relatively large nodes [1, 5], containing a variety of hardware, making the one-application per node model sub-optimal. CoApps enables workflows to operate in this newer environment by enhancing the SODA model to enable collocation of workflow components. To do this, the SODA framework had to be extended in three concrete ways. First we needed communication mechanisms to enable position independence so workflow components can operate in all definitions of in situ and have to provide strategies to avoid communication interference that arises during collocation. Second, we need better launching mechanisms than the coarse-grained batch schedulers found on current generation supercomputers. Finally, we need orchestration and monitoring constructs to discover and take advantage of collocation opportunities. In particular, we need to monitor platform utilization to identify spare resources. The first two extensions are discussed in Chapter 5.2.1 and the third in Chapter 5.2.2.

5.2.1 Communication Mechanisms and Launching

To achieve position independence, CoApps had to make several modifications to the SODA/Flexpath framework in regards to communication, both within a workflow component (i.e., MPI style communication) and across them.

Typically, supercomputer users rely on mechanisms such as “aprun” or “mpirun” to launch their codes, but many supercomputers have strict limitations such that only one application’s ranks/processes can be executing on a node. It is crucial to our formulation of the CoApp model that we be able consolidate multiple applications on a

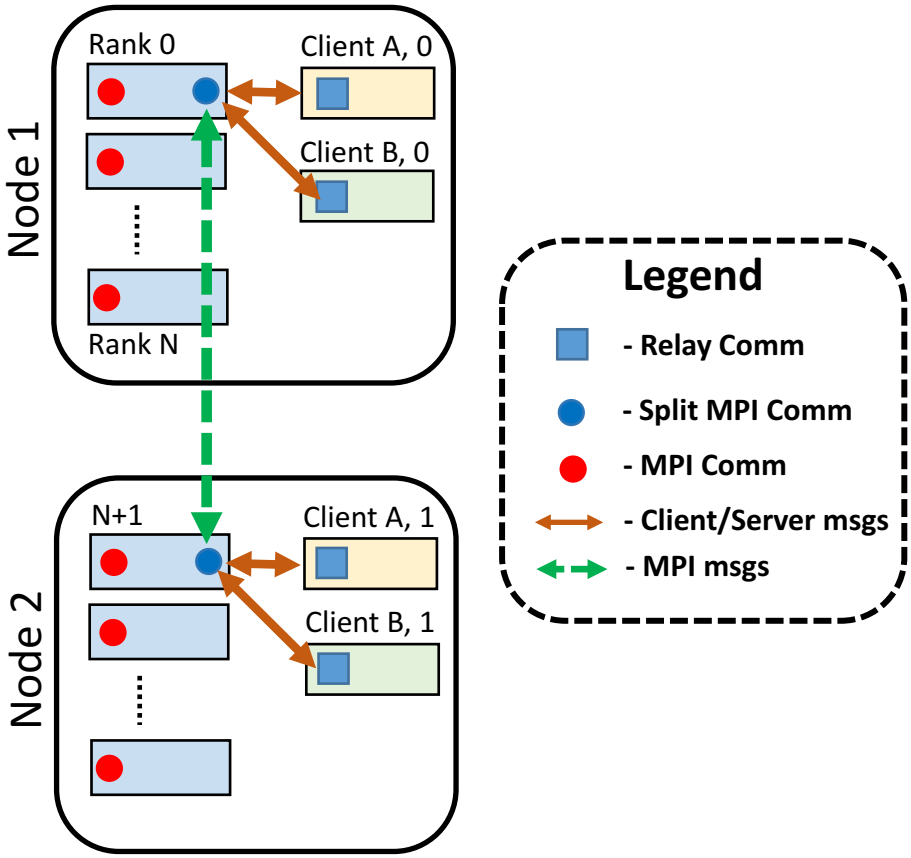


Figure 19: MPI Relay communication with processes launched with fork/exec

node. To get around these restrictions, we rely on standard fork/exec when launching codes for node-sharing, but we lose the forked application's ability to use a standard MPI communicator. When collocating codes on the same physical node, there are issues of added network traffic which may cause contention leading to severe performance penalties for the main simulation[9]. To address both the launch and network contention issues, we created MPIRelay, depicted in figure 19, which "intercepts" an application's MPI call (the MPIRelay_client) and relays it to the parent application owning a fully functioning MPI communicator (the MPIRelay_server), which then performs the MPI operation on behalf of the child process.

MPIRelay provides two modes of operation based on the multithreaded options MPI implementations offer. In the fully threaded mode (MPI_THREAD_MULTIPLE),

MPI allows for any thread to make calls on an MPI communicator, and if using one of those implementations, MPIRelay_server will create a separate thread to process the MPIRelay_client's requests. For MPI implementations that do not support multithreading, we queue the requests from the client, and provide a function, *process_mpi_requests*, that the user can periodically call that will process the requests on the queue.

The lowest MPI rank on each node operates as the MPIRelay_server and registers itself with their Workstation Orchestration from which it receives command messages specifying the application to fork/exec and its arguments. The server then spawns the child process sending to it the child's "MPI rank", the world size of the child application, and the server's communication end point. The child processes then call MPIRelay_init with these arguments, receive an MPIRelay communicator, and then proceed as normal.

To address communication interference, we use application-level hints, delivered via the ADIOS interface. Using hints, the higher-priority application can indicate when it is entering or leaving a communication-heavy phase. Upon entering, a flag is set and all communication from the child process is delayed until the flag is set back to off. When the flag is turned off, the client's MPI communications are then processed. The current implementation allows the child process's MPI communication to finish before subsequent communication is halted, so there will be some overlap, but our results show substantial improvement over the unmitigated case.

MPIRelay is implemented using the EVPath messaging system and exports a similar interface to that of MPI (i.e., instead of MPI_Bcast, it is MPIRelay_Bcast). The current implementation supports blocking calls and some non-blocking calls (MPI_Isend and MPI_Irecv). Future work will put a standard MPI interface on top of MPIRelay so applications will not have to make any code changes.

5.2.2 Monitoring and Control

The CoApps model is more robust than the initial SODA model for two reasons. First, it can enable collocation of workflow components. Second, it can enable per-node optimizations that SODA previously couldn't, such as communication interference, or even integrating with other interference management run times such as [67, 33]. To allow for our run time orchestration hierarchy to make decisions about collocation, we had to modify the SODA information bus to monitor and collect node-level information. In the current implementation, we use PAPI based monitoring for CUDA enabled GPUs and rely on polling the proc filesytem for CPU usage information.

The information bus collects this information at the end of every simulation output phase, within the ADIOS interface, and delivers it to the orchestration hierarchy, which keeps a history as well as the completion time for that epoch. Tracking completion time is important as from it, we can track progress and we can determine if our orchestration decision, i.e., deciding to collocate two components, is a bad one as it will cause the epoch completion time to increase.

This information is exposed to orchestrators through the SODA programmatic API and using this information, and the orchestration commands, users can create rules that check for conditions and then issue actions when these rules are met. One example that we demonstrate in Chapter 5.3 is that after collocation, we notice that the epoch completion time for the simulation starts to increase past a user defined threshold of 10%. The run time then chooses a different collocation strategy by collocating two analysis components together on a node separate from the core simulation.

While the current implementation extended the ADIOS API to include markers to indicate when applications are using the network, we also expect that hints can be used to indicate a wide variety of special conditions, such as when an application is entering an adaptive mesh-refinement phase. Enabling such functionality would allow

the run time to better understand behavior that deviates from the steady state.

5.3 Experimental Evaluation

The experiments are are evaluated on Oak Ridge National Laboratory's Titan machine, described in 4.4 as well as on Falcon, a local Linux cluster at Georgia Tech. Unlike the earlier experiments on Titan for this thesis, here we use Titan's NVIDIA Tesla K20 GPUs when making node consolidation options. Conversely, Falcon is an 80 node infiniband cluster with each node containing 12 cores and 24GB of RAM.

LAMMPS and Superglue are used to demonstrate CoApps ability to reason about performance and resource utilization and make collocation decisions accordingly. These experiments are run on Titan at scales ranging from 4096 processes to 16384 processes (one process per core).

For the overhead and interference management experiments, we use I/O kernel applications to stress the limits of MPIRelay and also to generate large amounts of interference as the LAMMPS + Superglue experiments do not generate significant communication interference. These experiments measuring communication overheads in comparison to standard MPI were conducted on Titan, and the evaluation of MPI Relay's interference management was conducted on Falcon in order to show that the approach is capable of operating on different classes of clusters.

5.3.1 Node Consolidation

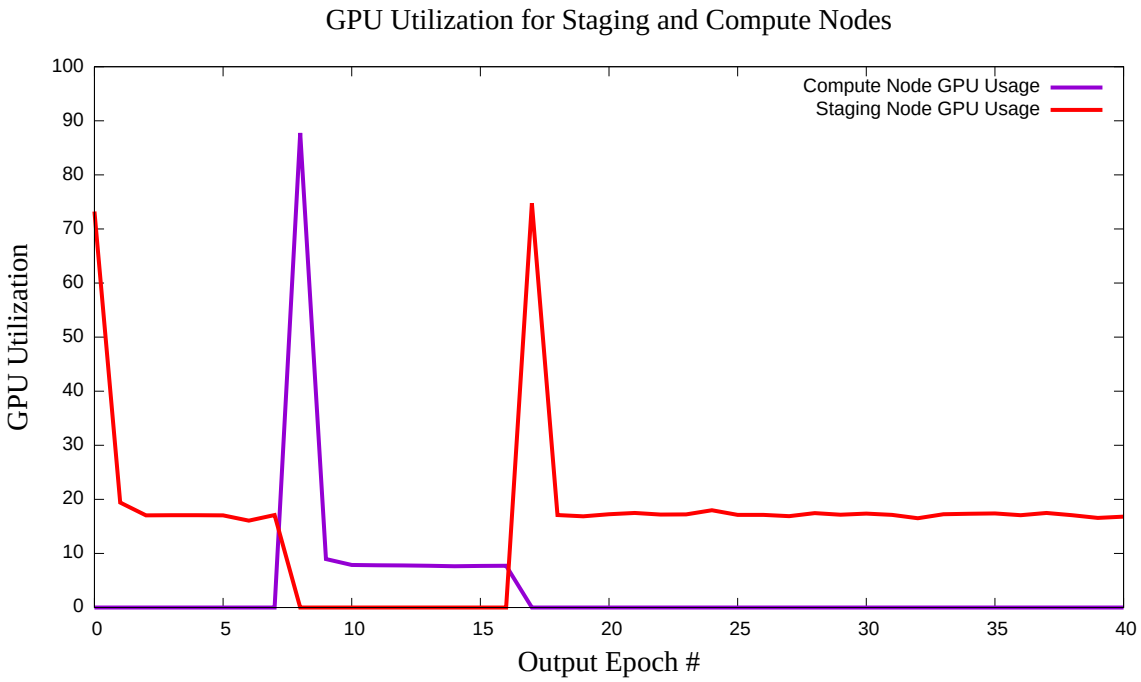
LAMMPS and Superglue is used to evaluate CoApps ability to make co-allocation decisions. In all instances, the workflow is initially deployed with LAMMPS executing at 16 processes per node and the workflow components (select, magnitude, and histogram) are deployed on separate nodes in the staging area at a ratio of 128 Lammps processes to one select and one magnitude process, and 64 Lammps processes to 1 histogram process. CoApps then monitors, on a per-epoch basis, the resource utilization and the time the components spend on processing and waiting for data.

Using the latter two metrics allows us to reason about each component's progress and performance.

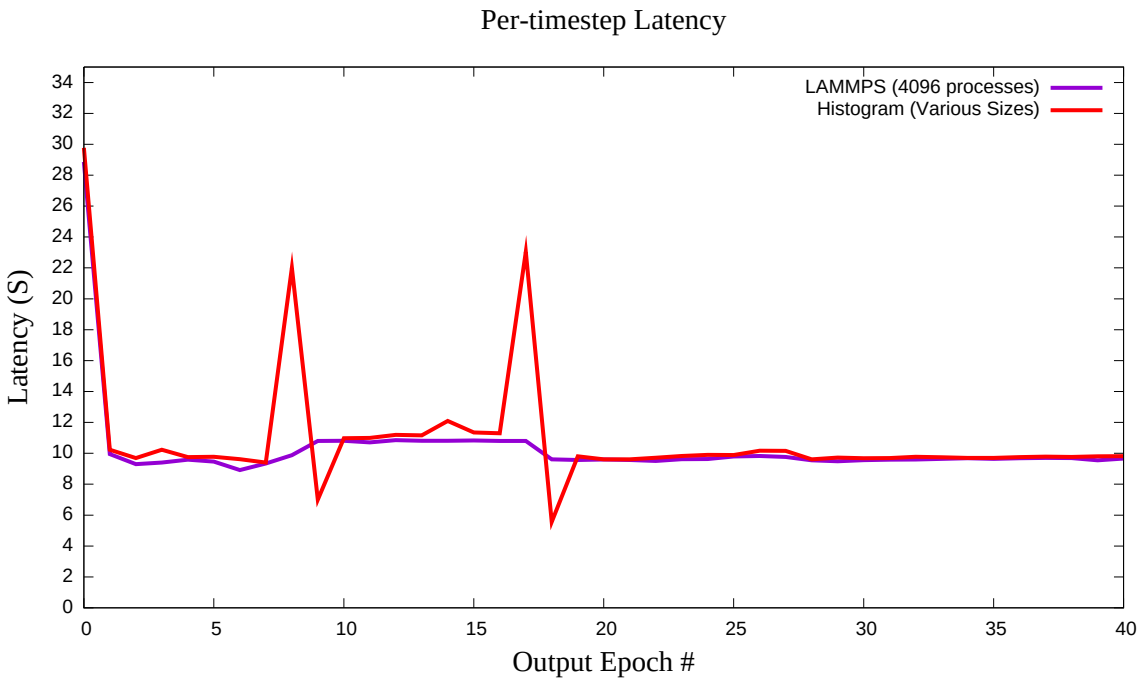
We use straight-forward orchestration rules to demonstrate the functionality and utility of the system; if the summation of resource usage between the two components does not exceed 100%, and if the throughput of the components (measured as epochs of data processed per minute) are similar within 5%, we attempt to do a consolidation. The rules used in these experiments also specify a threshold such that if we cause the simulation to slowdown by more than 5%, we undo the consolidation and try a different arrangement. While these rules might not work for all use cases, it is worth pointing out that using the programmatic API, one could create and try many different rules or even have smarter orchestration policies; the focus of this thesis isn't on the policies itself, but rather on the demonstration of management and agility at scale.

In these experiments, CoApps detects that the compute nodes have free GPU cycles and that the analysis components spend over 75% of their time waiting for data, so it decides to run all three components in situ on the compute nodes with LAMMPS.

Figures 20(a), 21(a), and 22(a) depict the GPU utilization during each epoch for the compute nodes and the staging nodes. The compute nodes initially have no GPU utilization as LAMMPS is running on only the CPU, while the staging area is executing the GPU enabled histogram. In each instance, when we move to the in situ case, the GPU usage on the compute nodes increases, but it is less than what it was in the staging area. This is because we have changed the ratio of histogram processes to Lammps processes; we moved from a ratio of 64:1 to 16:1. The large spikes in latency and GPU usage immediately following every reorganization happens each time the Thrust-GPU-enabled histogram makes the first call to the GPU, and this is repeatable when running these codes without the CoApps middleware.

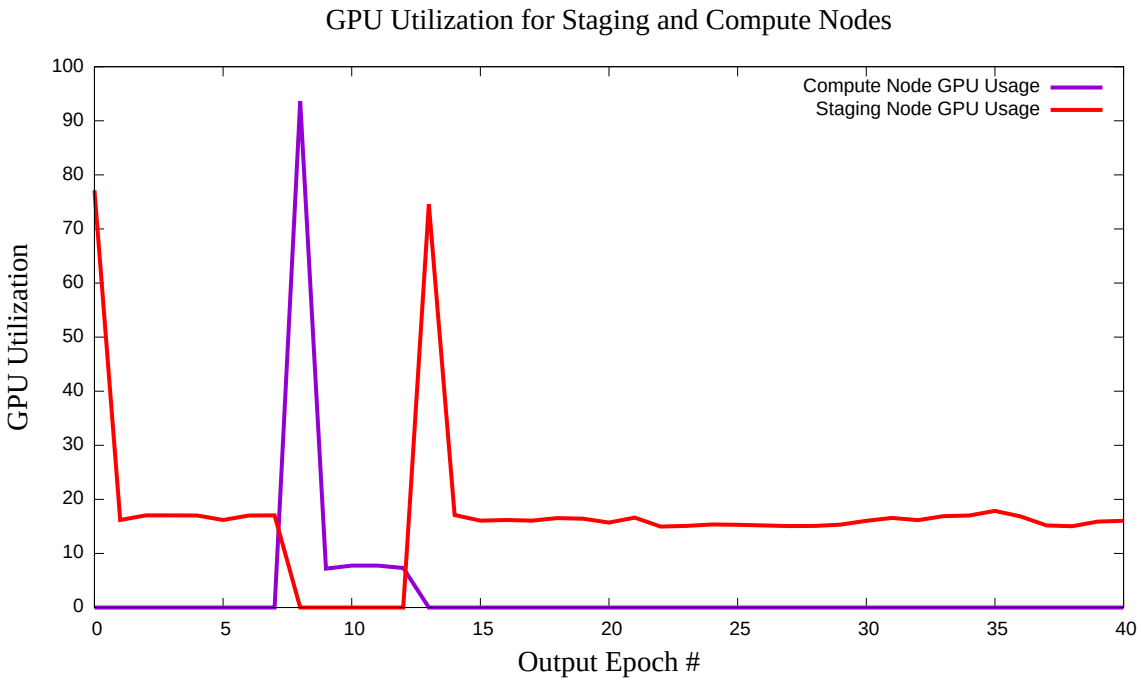


(a) GPU Utilization for Compute Nodes and Staging Nodes for Lammps + Superglue

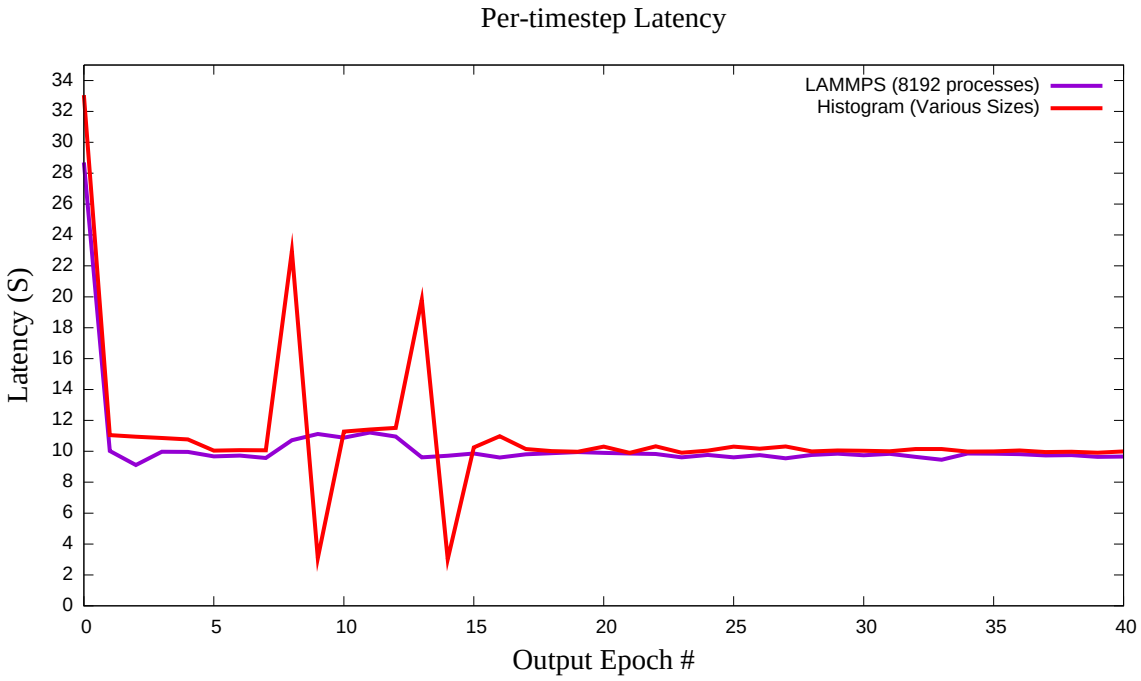


(b) Per-timestep latency for Lammps + Superglue

Figure 20: Node Consolidation (GPU utilization and component latency) at 4096 cores

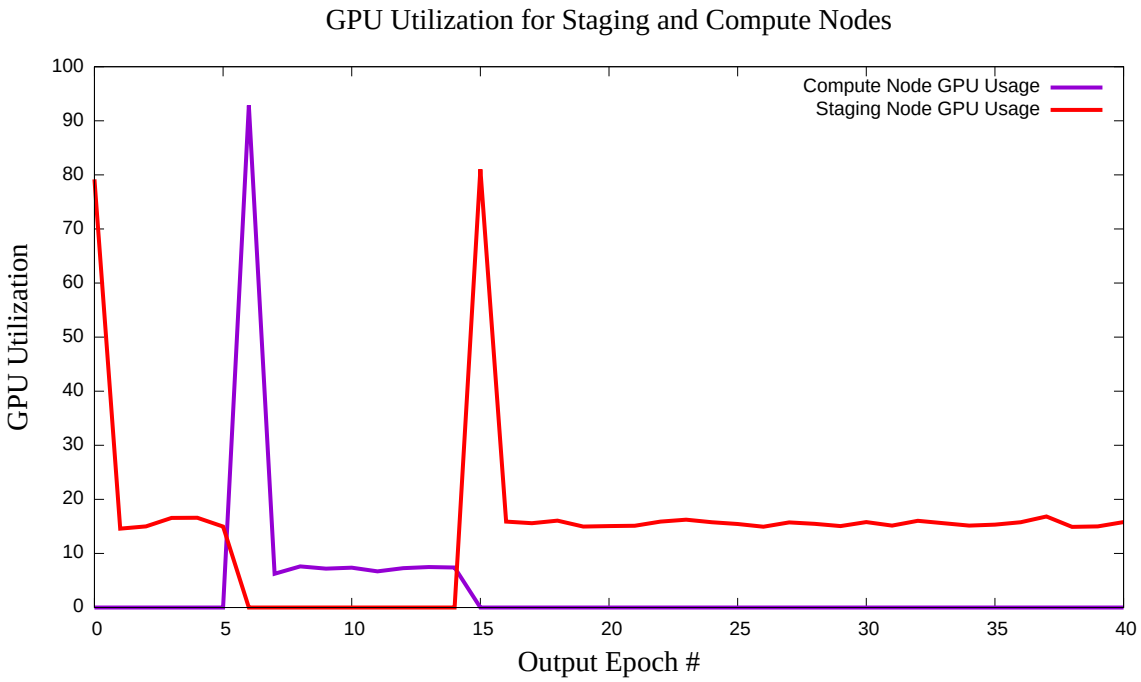


(a) GPU Utilization for Compute Nodes and Staging Nodes for Lammps + Superglue

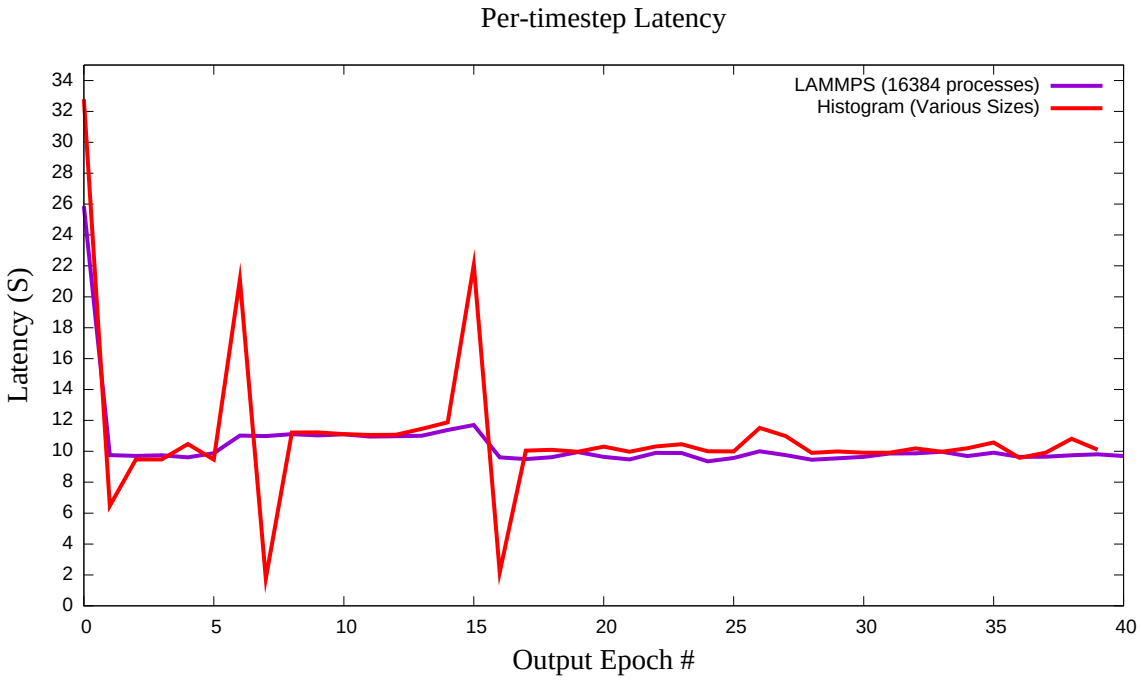


(b) Per-timestep latency for Lammps + Superglue

Figure 21: Node Consolidation (GPU utilization and component latency) at 8192 cores



(a) GPU Utilization for Compute Nodes and Staging Nodes for Lammps + Superglue



(b) Per-timestep latency for Lammps + Superglue

Figure 22: Node Consolidation (GPU utilization and component latency) at 16384 cores

Figures 20(b), 21(b), and 22(b) show the component latency, which is a measure of how long it takes the component to output an epoch of data. For graph clarity, we only show the latency for Lammps and for the histogram code; select and magnitude run in less than a second for each epoch of data. In all instances, we see that when collocating the analysis with the simulation, we generate interference between 10 and 15%, above our threshold of 5%. CoApps detects this slowdown on the simulation and then moves the analysis codes off of the simulation nodes onto a set of staging nodes, but keeping the three components collocated on the staging nodes instead of running on separate nodes. By doing so, we were able to run the same workflow, and keep the same level of throughput, while decreasing the number of staging nodes by 50%.

5.3.2 MPI Relay Evaluation

MPI Relay is evaluated in two parts. The first part, shown in figures 23, 24, and 25, compares the communication overhead of using MPI Relay and compares it with native MPI, and the second part (fig. 26 demonstrates the effectiveness of the interference management strategy.

Since MPI Relay ultimately relies on MPI, it is impossible for Relay to beat native MPI. The bulk of the additional costs of using MPI Relay are due to the round trip time of sending data to and from the simulation. In all cases though, the costs of using MPI Relay grow along with the costs of using MPI. While having additional overhead is less than ideal, we feel it is acceptable as the analysis components are often considered optional second class citizens compared to the sacrosanct simulation. The overheads may be improved with a more robust shared memory transport for EVPath.

To evaluate MPI Relay's interference management, we rely on using an I/O kernel that generates a lot of MPI traffic of mixed message sizes ranging from a few bytes

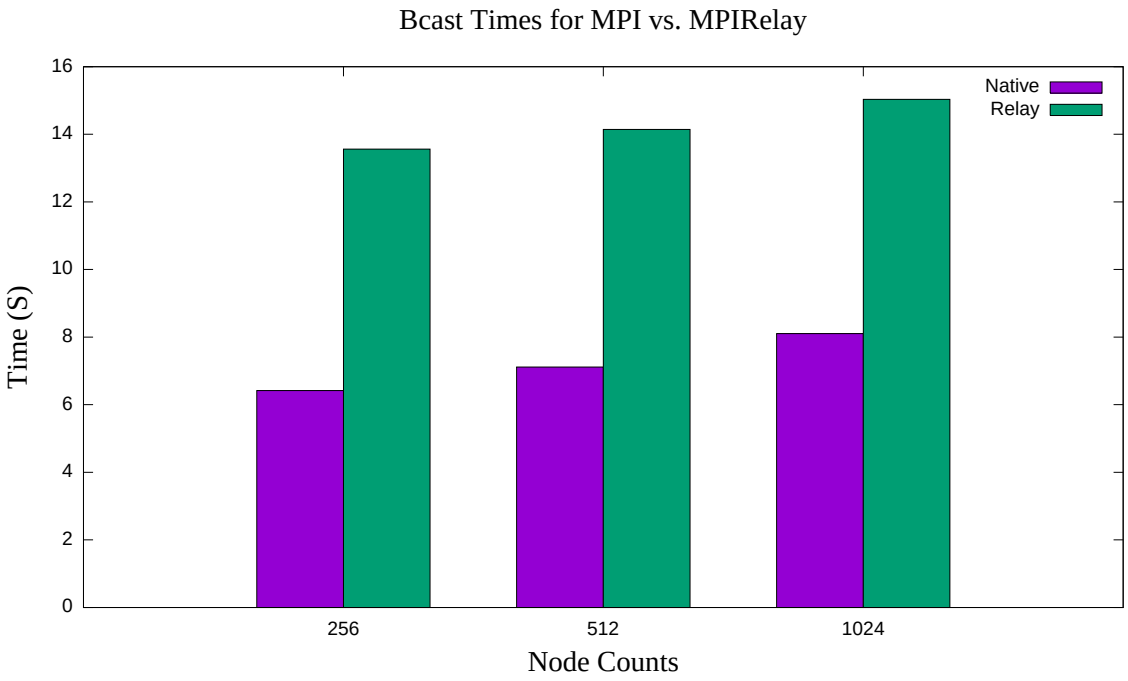


Figure 23: Comparison of Broadcast Times for MPI vs. MPI Relay transferring a 1GB buffer

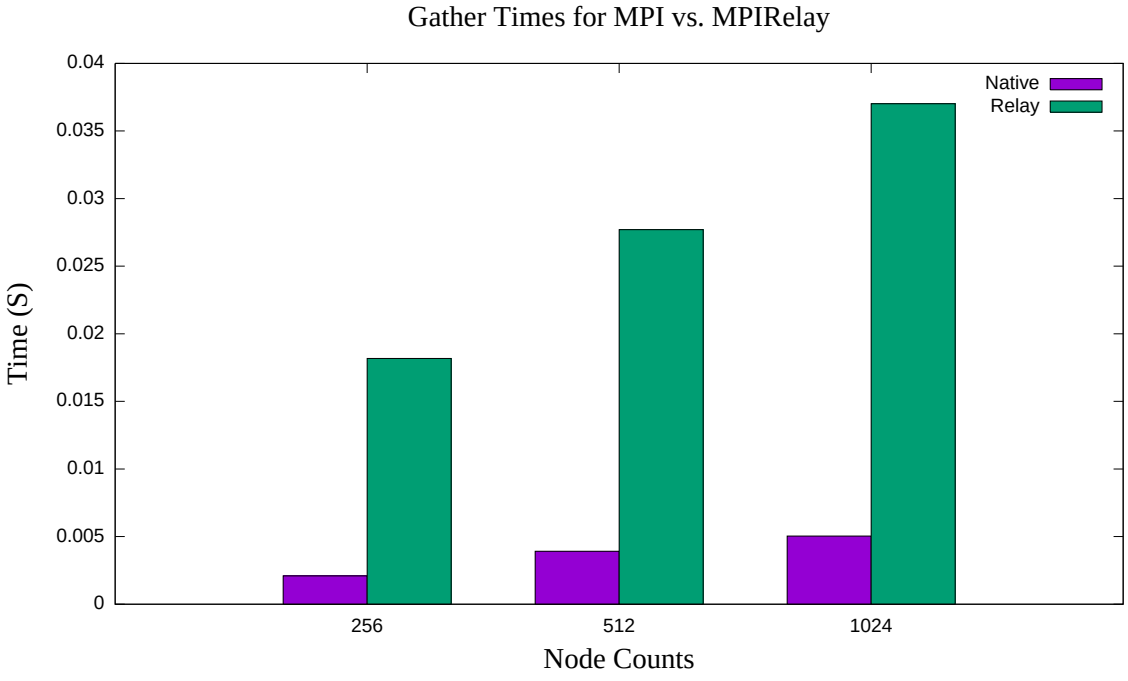


Figure 24: Comparison of Gather Times for MPI vs. MPI Relay tranferring 10KB buffers

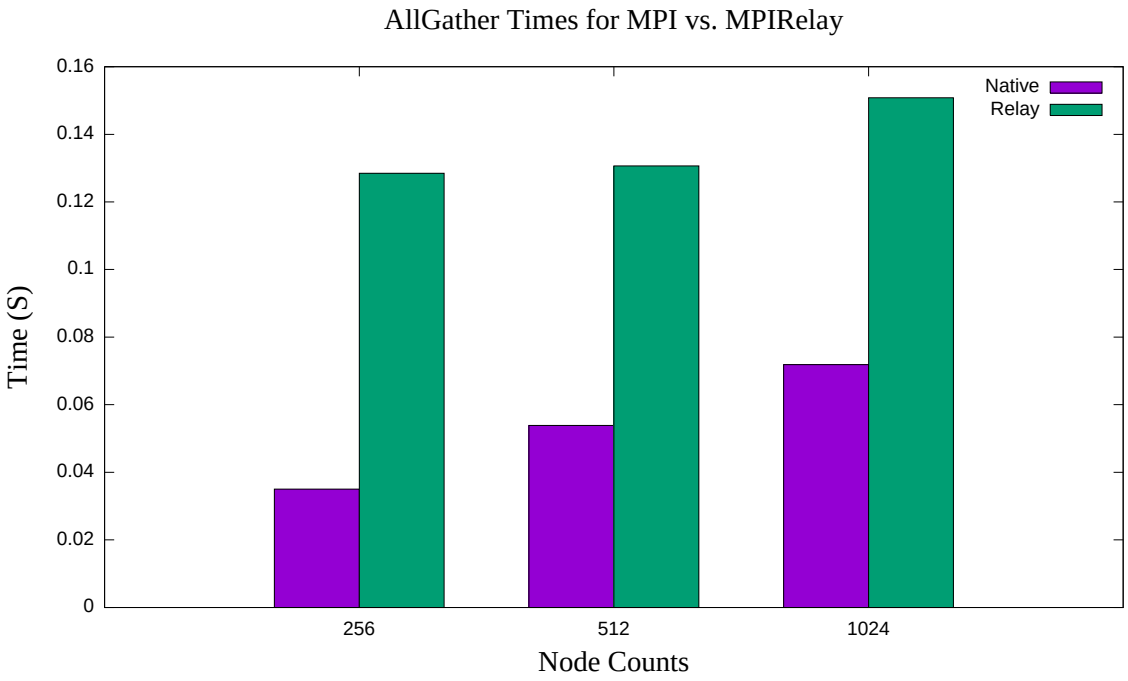


Figure 25: Comparison of All Gather Times for MPI vs. MPI Relay transferring 10KB buffers

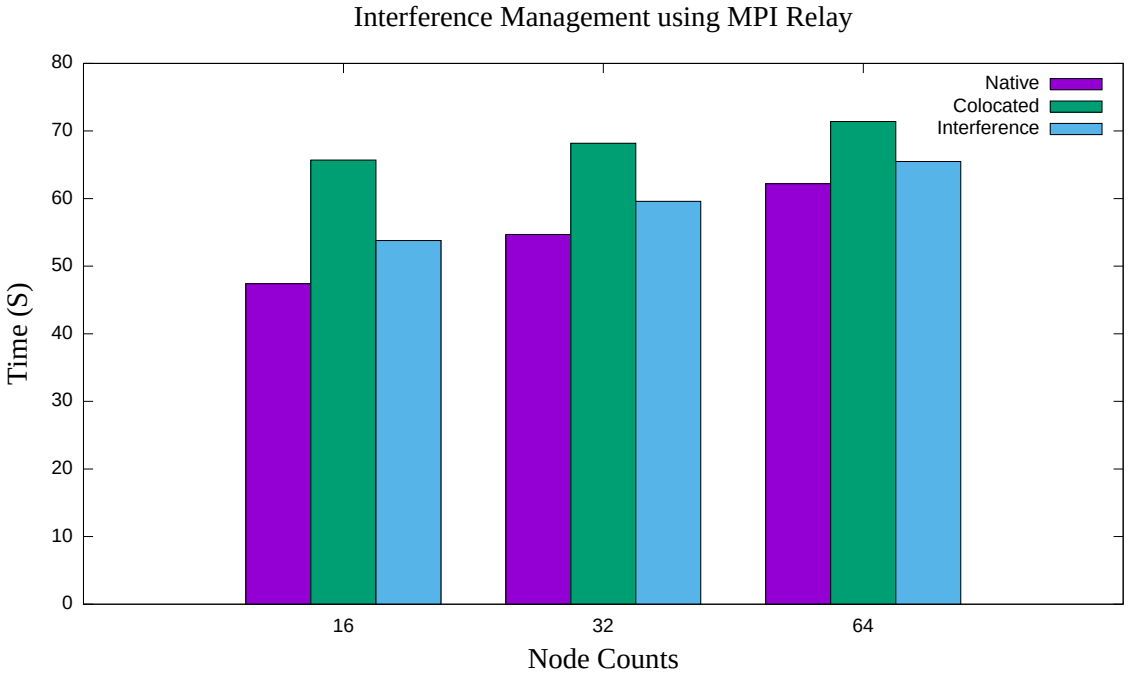


Figure 26: Interference Management using MPI Relay on University Linux Cluster

to 10MB. We run two instances of this I/O kernel, one serving as the high-priority application owning a full MPI communicator, and the other using MPI Relay. The goal of the interference management is to reduce the amount of perturbation on the high-priority application's communication. Figure 26 is structured as follows. Native is how much time collectively the I/O kernel spent on communicating when it was running by itself, with nothing running in situ. The bars labeled "Collocated" show how the communication time increases when we run another application in situ using MPI Relay, and finally, the bars titled "Interference" depict the communication time with interference management turned on. In all cases, we are able to reduce the overheads by over 50%. One reason for the remaining overhead is that we do not have any way to stop on-going MPI communication, and for transmitting large messages, this can be costly. Additionally, the traffic within a node may also cause problems as MPI and MPI Relay will both collide for intra-node communication.

5.3.3 Discussion

The results presented in this chapter demonstrate the utility of CoApps at both on enterprise class machines and on smaller university scale clusters. The benefits of CoApps is shown in in two parts: (1) its flexibility to make collocation decisions and detect and recover from "bad" collocation decisions due to interference or potentially other issues; and (2) manage communication interference during collocation. While in these experiments, collocating the analysis with the simulation induced a performance penalty, CoApps was able to collocate the analysis codes with each other and reduce the number of staging resources by 50%. To increase the viability for collocation, it would be possible to integrate other in situ management run times, such as [67, 33, 37, 55, 50] into our system, but we leave this for future work.

CHAPTER VI

RELATED WORK

6.0.1 Communication Mechanisms and Code Coupling

Scalable pub/sub implementations created outside the HPC domain tend to consider workloads comprised of large numbers of small, potentially unrelated, messages. BlueDove [43] from IBM is an attribute-based pub/sub implementation for elastic Cloud-based applications, intending to use the Cassandra data store. Since it deals with small messages, it is able to benefit from the routing of messages to external dispatcher servers that also perform subscription matching before delivering the messages to the subscribers. Flexpath differs in its focus on structured, potentially complex and voluminous data events transmitted between publishers and subscribers, with its consequent use of direct connections between both. This is also the case for [16, 41], which are pub/sub systems that aim to overcome some of the inefficiencies found with routing messages and subscriptions through processing overlay networks: publisher messages are first pushed to content brokers, and subscriber subscriptions are then routed through the network to find the correct overlay node that has matching data.

In the HPC space, the work presented in [38] outlines a content-based pub/sub infrastructure layered on top of the Dataspaces [25] substrate. The work allows for introspection into the data, and subscribers can register to receive sub-samples of the events based on avg/min/max values computed from the data while it is in-flight. Flexpath differs by (1) using a direct connect model to avoid the extra data movements involved with publishing data to an external broker, as in the shared-space abstraction offered by Dataspaces; and (2) offering a subscription model that can go beyond standard array-slicing and chunking to allow publishers and subscribers to

produce and consume complex data, including graphs or arrays of complex types; it also permits codes that may have type-mismatches between publishers and subscribers to exchange data.

6.0.2 Orchestration, Big Data Systems, and Workflow Management

Previous work on datacenter management and for "big data" systems uses techniques like elasticity and replication to provide scalability and fault tolerance [35, 49, 65, 6], but do not address directly the end-to-end behaviors and resource restrictions of the parallel analytics pipelines SODA/CoApps manages. Specifically, with the CoApps model, we can realize the diverse orchestration semantics needed for such pipelines expressed with SLAs and drive orchestration actions that implement the limited types of elasticity permitted by the HPC machine, the degree of reactivity needed for effective pipeline use, and the desired end-to-end behaviors, such as throughput or latency.

While CoApps may appear like a limited hypervisor, it is not concerned with node-level partitioning and running multiple entities with performance and security isolation. SODA workstations are more akin to "resource islands" explored for high end multicore processors [57]. They differ in the explicit orchestration policies and actions specification and in enabling custom and application-specific methods for managing analytics pipelines.

Other HPC-centric work on managing analytics and visualization pipelines [37] provides adaptation policies at different stack layers (cross-layer adaptation) targeting an adaptive mesh refinement (AMR) code. It focuses on specific policies at different layers, to ensure minimal time to solution, whereas our work investigates the mechanics and abstractions of management that would be suitable for analytics pipelines; the policies discussed in [37] are examples of additional policies suitable for implementation with the CoApps framework.

Initial results [24] demonstrate some of the concepts discussed here, but the work presented in this paper (1) extends upon the model and orchestration constructs, (2) explores a wider variety of use cases, including an understanding of how state and metadata are managed (i.e., quality of data and fault recovery), (3) describes how SLAs are defined and how policies are constructed to enforce them, and (4) extends the concepts to pipeline that span multiple machines by leveraging the Flexpath [22] staging solution operating across a variety of interconnects. Our earlier solution was implemented with the Cray Portals API [15] and only operated on high end machines.

CHAPTER VII

FUTURE WORK

The CoApps abstraction provides a significant improvement over traditional analytics methods for large scale science applications. There are still, however, many directions in which the work in this thesis can be augmented and advanced. In particular, CoApps requires additional efforts in addressing a wider array of hardware expected to be found on next generation platforms such as other accelerators, burst buffers, and NVRam technologies. Additionally widescale adoption of new technologies is highly dependent on programmability and CoApps should take advantage of newer programming models [12] and workflow construction languages such as Swift [60].

7.1 Utilizing New Hardware Technologies

Another hot area of research in this space is burst buffer technology. The primary goal of this technology is to shield applications from the performance losses when writing to global storage systems. Instead, data is captured on node-local storage devices before eventually making its way to global storage. Following on this, "active burst buffers" seek to provide the types of in situ analysis we discussed in this thesis with some added persistence benefits in the data path. Indeed our communication mechanisms could be augmented to use NVRam or burst buffers to provide greater functionality such as out-of-core analysis support. We should also leverage the persistence features of these technologies to explore better ways at addressing fault tolerance.

There are a number of other run times that aim to manage the lower level hardware to provide features such as multi-tenancy [33] or manage interference [67]. Following on the results presented in this thesis, integrating other these other run times into this framework would provide greater in situ and colocation opportunities.

7.2 Programmability and Usability

The scope of this thesis did not include an in-depth discussion of the programming model for workflow construction and management. On-going work relating to partitioned global address space (PGAS) models for code coupling is an important model to consider given that recent work has explored how PGAS can leverage deep-memory hierarchies for persistence and other properties.

Domain specific languages like Swift and Legion are also of interest. Currently, CoApps makes use of shell scripts and Linux system calls for launching and reorganizing workflow components, but languages like Swift provide a much more natural and programmatic way of describing and executing workflows.

Parallel programming models like Legion and OCR also provide additional parallel computing models not made readily available using MPI such as task-based paralellism. Supporting different models for parallelism fits into the CoApps model as a key goal of the orchestration hierarchy is to allow for customized management.

CHAPTER VIII

CONCLUSION

The immense data volumes have become a significant performance bottleneck for high performance scientific applications and dealing with them requires a paradigm shift in both the applications and the platforms running them. In this thesis, we detail our vision on how we can bridge the gap between these two orthogonal paradigm shits using the CoApps model for in situ analytics. CoApps combine low-overhead data movement using the Flexpath code coupling transport, coarse grain orchestration via the SODA framework, and fine-grained colocation and resource management using the CoApps run time, to enable a wide variety of management and re-organization opportunities at run time.

The use of CoApps extends the locations where analytics functions can execute by providing end users with a way to write their code generically, via the ADIOS api, and letting the orchestration hierarchy determine an optimal placement to meet their wide ranging end goals. Using MPIRelay, end users can also avoid technical/administrative limitations when using in situ analytics on high-end machines and can also ensure healthy performance by taking advantage of the communication interference it offers.

In summary, CoApps has been demonstrated to be a viable piece of the solution towards scalable in situ analytics for the next generation of leadership class machines and the applications that use them. The combination of efficient code coupling, location independent analytics components, and colocation are all brought together in providing new functionality to large scale science workflow. This functionality has been demonstrated to go beyond static workflows relying on resource overprovisioning, to offering end users with a more functional way to use exascale resources. Parts of

this research have already been adopted as part of the official ADIOS release package and has been presented as live demos at key conferences, such as Supercomputing.

REFERENCES

[1] "Coral supercomputers."

[2] "Gts gyrokinetic pic code."

[3] "Hadoop: http://hadoop.apache.org."

[4] "Lxc-linux containers."

[5] "Oak ridge national labs summit machine."

[6] "Storm: Distributed and fault-tolerant realtime computation."

[7] *IEEE/ACM 16th International Symposium on Cluster, Cloud and Grid Computing, CCGrid 2016, Cartagena, Colombia, May 16-19, 2016*, IEEE Computer Society, 2016.

[8] "Superglue: Standardizing glue components for hpc workflows," September 2016.

[9] Abbasi, H., Wolf, M., Eisenhauer, G., Klasky, S., Schwan, K., and Zheng, F., "DataStager: scalable data staging services for petascale applications," *Cluster Computing*, vol. 13, pp. 277–290, 2010. 10.1007/s10586-010-0135-6.

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

[11] AbdelBaky, M., Montes, J. D., Parashar, M., Unuvar, M., and Steinder, M., "Docker containers across multiple clouds and data centers," in *8th IEEE/ACM International Conference on Utility and Cloud Computing, UCC 2015, Limassol, Cyprus, December 7-10, 2015* (Raicu, I., Rana, O. F., and Buyya, R., eds.), pp. 368–371, IEEE Computer Society, 2015.

[12] Bauer, M., Treichler, S., Slaughter, E., and Aiken, A., "Legion: expressing locality and independence with logical regions," in *SC Conference on High Performance Computing Networking, Storage and Analysis, SC '12, Salt Lake City, UT, USA - November 11 - 15, 2012* (Hollingsworth, J. K., ed.), p. 66, IEEE/ACM, 2012.

[13] Borgdorff, J. and et. al, "Distributed multiscale computing with MUSCLE 2, the multiscale coupling library and environment," *CoRR*, vol. abs/1311.5740, 2013.

[14] Boyuka, D. and et al, "Transparent in situ data transformations in adios," in *CCGrid*, pp. 256–266, May 2014.

[15] Brightwell, R. and et. al, "Implementation and performance of portals 3.3 on the cray XT3," in *Cluster*, pp. 1–10, 2005.

[16] Carzaniga, A., Rutherford, M., and Wolf, A., "A Routing Scheme for Content-Based Networking," in *INFOCOM 2004. Twenty-third AnnualJoint Conference of the IEEE Computer and Communications Societies*, vol. 2, pp. 918–928 vol.2, 2004.

[17] Castro, M. and Liskov, B., "Practical byzantine fault tolerance," in *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, OSDI '99, (Berkeley, CA, USA), pp. 173–186, USENIX Association, 1999.

[18] Cedilnik, A., Geveci, B., Moreland, K., Ahrens, J. P., and Favre, J. M., "Remote Large Data Visualization in the ParaView Framework," in *EGPGV*, pp. 163–170, 2006.

[19] Chandrasekar, K., Seshasayee, B., Gavrilovska, A., and Schwan, K., "Task characterization-driven scheduling of multiple applications in a task-based runtime," in *Proceedings of the First International Workshop on Extreme Scale Programming Models and Middleware, ESPM 2015, Austin, Texas, USA, November 15, 2015* (Panda, D. K., Schulz, K. W., Hamidouche, K., and Subramoni, H., eds.), pp. 52–55, ACM, 2015.

[20] Childs, H., Duchaineau, M. A., and Ma, K.-L., "A Scalable, Hybrid Scheme for Volume Rendering Massive Data Sets," in *EGPGV* (Heirich, A., Raffin, B., and dos Santos, L. P. P., eds.), pp. 153–161, Eurographics Association, 2006.

[21] Childs, H., Geveci, B., Schroeder, W. J., Meredith, J. S., Moreland, K., Sewell, C., Kuhlen, T., and Bethel, E. W., "Research Challenges for Visualization Software," *IEEE Computer*, vol. 46, no. 5, pp. 34–42, 2013.

[22] Dayal, J., Bratcher, D., Eisenhauer, G., Schwan, K., Wolf, M., Zhang, X., Abbasi, H., Klasky, S., and Podhorszki, N., "Flexpath: Type-based publish/subscribe system for large-scale science analytics," in *14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGrid 2014, Chicago, IL, USA, May 26-29, 2014*, pp. 246–255, IEEE Computer Society, 2014.

[23] Dayal, J., Cao, J., Eisenhauer, G., Schwan, K., Wolf, M., Zheng, F., Abbasi, H., Klasky, S., Podhorszki, N., and y Lofstead, J., "I/o containers: Managing the data analytics and visualization pipelines of high end codes," in *International Workshop on High Performance Data Intensive Computing*, 2013.

[24] Dayal, J. and et. al, "I/O containers: Managing the data analytics and visualization pipelines of high end codes," in *HPDIC '13*, pp. 2015–2024.

[25] Docan, C., Parashar, M., and Klasky, S., "DataSpaces: an interaction and coordination framework for coupled simulation workflows," in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, (New York, NY, USA), pp. 25–36, ACM, 2010.

[26] Edwards, H. C., Trott, C. R., and Sunderland, D., "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *J. Parallel Distrib. Comput.*, vol. 74, no. 12, pp. 3202–3216, 2014.

[27] Eisenhauer, G., Wolf, M., Abbasi, H., Klasky, S., and Schwan, K., "A Type System for High Performance Communication and Computation," in *e-Science Workshops (eScienceW), 2011 IEEE Seventh International Conference on*, pp. 183–190, 2011.

[28] Eisenhauer, G. and et. al, "Event-based systems: opportunities and challenges at exascale," in *DEBS*, 2009.

[29] Eugster, P., "Type-based Publish/Subscribe: Concepts and Experiences," *ACM Trans. Program. Lang. Syst.*, vol. 29, no. 1, 2007.

[30] Felter, W., Ferreira, A., Rajamony, R., and Rubio, J., "An updated performance comparison of virtual machines and linux containers," in *2015 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2015, Philadelphia, PA, USA, March 29-31, 2015*, pp. 171–172, IEEE Computer Society, 2015.

[31] Ferreira, K. and et. al, "Evaluating the viability of process replication reliability for exascale systems," in *SC '11*, pp. 44:1–44:12, ACM, 2011.

[32] Gerofi, B., Takagi, M., Ishikawa, Y., Riesen, R., Powers, E., and Wisniewski, R. W., "Exploring the design space of combining linux with lightweight kernels for extreme scale computing," in Hoefler and Iskra [36], pp. 5:1–5:8.

[33] Goswami, A. and et. al, "Landrush: Rethinking in-situ analysis for gpgpu workflows," in *CCGrid '16*, IEEE, 2016.

[34] Hawkes, E. R. and et. al, "Direct Numerical Simulation of Turbulent Combustion: Fundamental Insights Towards Predictive Models," *Journal of Physics: Conference Series*, vol. 16, no. 1, p. 65, 2005.

[35] Hindman, B. and et. al, "Mesos: a platform for fine-grained resource sharing in the data center," in *NSDI '11*, (Berkeley, CA, USA), pp. 22–22, USENIX Association.

[36] Hoefler, T. and Iskra, K., eds., *Proceedings of the 5th International Workshop on Runtime and Operating Systems for Supercomputers, ROSS 2015, Portland, OR, USA, June 16, 2015*, ACM, 2015.

[37] Jin, T. and et. al, "Using cross-layer adaptations for dynamic data management in large scale coupled scientific workflows," in *SC '13*.

[38] Jin, T., Zhang, F., Parashar, M., Klasky, S., Podhorszki, N., and Abbasi, H., "A Scalable Messaging System for Accelerating Discovery from Large Scale Scientific Simulations," in *HiPC*, pp. 1–10, IEEE, 2012.

[39] Lakshminarasimhan, S., Shah, N., Ethier, S., Klasky, S., Latham, R., Ross, R. B., and Samatova, N. F., "Compressing the Incompressible with ISABELA: In-situ Reduction of Spatio-temporal Data," in *Euro-Par (1)*, pp. 366–379, 2011.

[40] Leibiusky, J., Eisbruch, G., and Simonassi, D., *Getting Started with Storm - Continuous Streaming Computation with Twitter's Cluster Technology*. O'Reilly, 2012.

[41] Li, G., Hou, S., and Jacobsen, H.-A., "A Unified Approach to Routing, Covering and Merging in Publish/Subscribe Systems Based on Modified Binary Decision Diagrams," in *Distributed Computing Systems, 2005. ICDCS 2005. Proceedings. 25th IEEE International Conference on*, pp. 447–457, 2005.

[42] Li, J., keng Liao, W., Choudhary, A. N., Ross, R. B., Thakur, R., Gropp, W., Latham, R., Siegel, A., Gallagher, B., and Zingale, M., "Parallel netCDF: A High-Performance Scientific I/O Interface," in *SC*, p. 39, 2003.

[43] Li, M., Ye, F., Kim, M., Chen, H., and Lei, H., "A Scalable and Elastic Publish/Subscribe Service," *Parallel and Distributed Processing Symposium, International*, vol. 0, pp. 1254–1265, 2011.

[44] Lin, Z., Hahm, T. S., Lee, W. W., Tang, W. M., and White, R. B., "Turbulent transport reduction by zonal flows: Massively parallel simulations," *Science*, vol. 281, no. 5384, pp. 1835–1837, 1998.

[45] Lofstead, J. and et. al, "Adaptable, metadata rich io methods for portable high performance io," in *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pp. 1–10, May 2009.

[46] Lofstead, J., Dayal, J., Schwan, K., and Oldfield, R., "D2T: Doubly Distributed Transactions for High Performance and Distributed Computing," *Cluster Computing: To Appear*, 2012.

[47] Lofstead, J., Oldfield, R., and Kordenbrock, T., "Unconventional Data Staging Using NSSI," in *In Proceedings of IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, (Delft, The Netherlands), May 2013.

[48] 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: Research Articles," *Concurr. Comput. : Pract. Exper.*, vol. 18, pp. 1039–1065, August 2006.

[49] Moise, D. and et. al, "Improving the hadoop map/reduce framework to support concurrent appends through the blobseer blob management system," HPDC '10, pp. 834–840, ACM, 2010.

[50] Mondragon, O. H., Bridges, P. G., Levy, S., Ferreira, K. B., and Widener, P. M., "Scheduling in-situ analytics in next-generation applications," in *IEEE/ACM 16th International Symposium on Cluster, Cloud and Grid Computing, CCGrid 2016, Cartagena, Colombia, May 16-19, 2016* [7], pp. 102–105.

[51] Nasgaard, H., Gedik, B., Komor, M., and Mendell, M. P., "IBM InfoSphere Streams: Event Processing For a Smarter Planet," in *CASCON* (Martin, P., Kark, A. W., and Stewart, D. A., eds.), pp. 311–313, ACM, 2009.

[52] Peng, D. and et. al, "Large-scale incremental processing using distributed transactions and notifications," in *9th USENIX Symposium on Operating Systems Design and Implementation*, pp. 4–6.

[53] Plimpton, S. and et. al, "Particle-Mesh Ewald and rRESPA for Parallel Molecular Dynamics Simulations," in *PPSC*, SIAM, 1997.

[54] Riesen, R., Maccabe, A. B., Gerofi, B., Lombard, D. N., Lange, J. J., Pedretti, K. T., Ferreira, K. B., Lang, M., Keppel, P., Wisniewski, R. W., Brightwell, R., Inglett, T., Park, Y., and Ishikawa, Y., "What is a lightweight kernel?," in Hoefler and Iskra [36], pp. 9:1–9:8.

[55] Rodero, I., Parashar, M., Landge, A. G., Kumar, S., Pascucci, V., and Bremer, P., "Evaluation of in-situ analysis strategies at scale for power efficiency and scalability," in *IEEE/ACM 16th International Symposium on Cluster, Cloud and Grid Computing, CCGrid 2016, Cartagena, Colombia, May 16-19, 2016* [7], pp. 156–164.

[56] Sahoo, S. K. and Agrawal, G., "Supporting XML Based High-Level Abstractions on HDF5 Datasets: A Case Study in Automatic Data Virtualization," in *LCPC* (Eigenmann, R., Li, Z., and Midkiff, S. P., eds.), vol. 3602 of *Lecture Notes in Computer Science*, pp. 299–318, Springer, 2004.

[57] Tembey, P. and et. al, "intune: Coordinating multicore islands to achieve global policy objectives," TRIOS '13, (New York, NY, USA), pp. 4:1–4:16, ACM, 2013.

[58] Vishwanath, V. and et. al., "Toward simulation-time data analysis and i/o acceleration on leadership-class systems," in *Large Data Analysis and Visualization (LDAV), 2011 IEEE Symposium on*, pp. 9–14, Oct 2011.

[59] Wang, C. and et al, "A flexible architecture integrating monitoring and analytics for managing large-scale data centers," in *ICAC '11*, pp. 141–150, 2011.

[60] Wilde, M., Hategan, M., Wozniak, J. M., Clifford, B., Katz, D. S., and Foster, I., "Swift: A language for distributed parallel scripting," *Parallel Computing*, vol. 37, no. 9, pp. 633 – 652, 2011. Emerging Programming Paradigms for Large-Scale Scientific Computing.

[61] Wolf, M., Cai, Z., Huang, W., and Schwan, K., "Smartpointers: Personalized scientific data portals in your hand," in *Supercomputing, ACM/IEEE 2002 Conference*, pp. 20–20, Nov 2002.

[62] Wolf, M., Abbasi, H., Collins, B., Spain, D., and Schwan, K., "Service Augmentation for High End Interactive Data Services," in *CLUSTER*, pp. 1–11, IEEE, 2005.

[63] Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M. J., Shenker, S., and Stoica, I., "Resilient Sistributed Datasets: a Fault-Tolerant Abstraction for In-Memory Cluster Computing," in *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, NSDI'12, (Berkeley, CA, USA), pp. 2–2, USENIX Association, 2012.

[64] Zaharia, M. and et. al, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *NSDI 12*, (San Jose, CA), pp. 15–28, USENIX.

[65] Zaharia, M. and et. al, "Spark: Cluster computing with working sets," in *2nd USENIX Workshop on Hot Topics in Cloud Computing, HotCloud'10, Boston, MA, USA, June 22, 2010*, 2010.

[66] Zheng, F., Abbasi, H., Docan, C., Lofstead, J., Klasky, S., Liu, Q., Parashar, M., Podhorszki, N., Schwan, K., and Wolf, M., "PreDatA-Preparatory Data Analytics on Peta-Scale Machines."

[67] Zheng, F. and et al, "Goldrush: Resource efficient in situ scientific data analytics using fine-grained interference aware execution," in *SC '13'*, pp. 78:1–78:12, ACM, 2013.