DATA SERVICES: BRINGING I/O PROCESSING TO PETASCALE

A Thesis
Presented to
The Academic Faculty

by

Mohammad H. Abbasi

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
College of Computing

Georgia Institute of Technology
August 2011

DATA SERVICES: BRINGING I/O PROCESSING TO PETASCALE

Approved by:

Professor Karsten Schwan, Advisor
College of Computing
Georgia Institute of Technology

Professor Rich Vuduc
College of Computing
Georgia Institute of Technology

Dr Matthew Wolf
College of Computing
Georgia Institute of Technology

Dr Ron Oldfield

Sandia National Laboratories

Dr Scott Klasky

Oak Ridge National Laboratory

Date Approved: 7 July 2011

To my best friend Mariana Agha,

she always listened and never hung up.

To my mom,

my other best friend.

To my dad,

for always being supporting and never telling me to hurry up.

To Mac and Sammy,

both of you will be missed.

ACKNOWLEDGEMENTS

In addition to those in the dedication, I would like to thank my advisor, Karsten Schwan. Through all the ups and downs of my doctoral career Karsten has always provided invaluable help and support.

Additionally I'd like to give special thanks to my mentor and friend Matthew Wolf. He introduced me to research in high performance computing and helped focus my ideas through his knowledge of science. Similarly I owe a large debt of gratitude to both Greg and Peggy Eisenhauer. Greg for providing not just technical help but moral support throughout the years and Peggy for always letting me stay at her house and making me feel at home. My last few years would have been unbearably dull without them. Thanks also to Scott Klasky, who has given me invaluable guidance in converting the abstract ideas behind data services into concrete usable design for the infrastructure.

My fellow students, Jay Lofstead and Fang Zheng have provided both advice and direction, but in the case of Fang, also helped me perform many of the experiments and served as an early user of data services and helped me debug a lot of this infrastructure.

I'd also like to acknowledge the Refugee Family Services and the International Community School for letting me be a part of their efforts to provide help and education for refugees living in Atlanta. Even when things were terrible I knew that the little boys and girls I worked with would put a smile on my face.

Finally I'd like to thank four close friends. Mariana Agha, mentioned in the dedication, my best friend. She was always there to lend an ear and give friendly advice. Irfan Ali Khan, my oldest friend, whose own Ph.D. gave me even more motivation to finish myself. And Shalini and Jyoti Gupta, whose presence in Atlanta made the city brighter.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

The increasing size of high performance computing systems and the associated increase in the volume of generated data, has resulted in an I/O bottleneck for these applications. This bottleneck is further exacerbated by the imbalance in the growth of processing capability compared to storage capability, due mainly to the power and cost requirements of scaling the storage. This thesis introduces data services, a new abstraction which provides significant benefits for data intensive applications. Data services combine low overhead data movement with flexible placement of data manipulation operations, to address the I/O challenges of leadership class scientific applications. The impact of asynchronous data movement on application runtime is minimized by utilizing novel server side data movement schedulers to avoid contention related jitter in application communication. Additionally, the JITStager component is presented. Utilizing dynamic code generation and flexible code placement, the JITStager allows data services to be executed as a pipeline extending from the application to storage. It is shown in this thesis that data services can add new functionality to the application without having an significant negative impact on performance.

CHAPTER I

INTRODUCTION

The last half decade has seen a dramatic increase in both the scale and importance of high performance computing. With the increase computational capability of leadership class machines has come a large increase in the data produced and consumed by leadership applications. Typical data rates for applications have increased from terabytes/day to close to petabytes/day, with another order of magnitude increase in data size looming on the horizon. However the increased data sizes have not been accompanied by comparable performance increase in the storage systems available to the largest machines, with both available data size and total throughput seeing limited increases over this timeframe.

For high end machines, this imbalance between the I/O subsystems performance and the load on the I/O system has resulted in a significant bottleneck in fully exploiting the performance of current generation machines, and it will play an even large role in placing limits on the efficient utilization of future systems.

1.1 The I/O imbalance

In today's competitive research environment, high performance computing plays a significant role in driving innovation in a multitude of fields, including in cutting edge research for nuclear fusion, combustion modelling, climate prediction, and materials development. Scalable computing has been a driver for innovation in these fields, but as high performance computing approaches petascale, the cost of I/O becomes significant. Current generation applications are already producing data volumes approaching 16 TB/output step, but the cost of the I/O operation is not the only overhead associated with such large volumes of data. Analyzing the data to gain scientific understanding, finding features within the data set and producing meaningful results from these steps are inherent parts of the scientific

data workflow. Doing so at scale is inordinately difficult, for reasons explained in more detail next. Before doing so, we note, however, that these issues are not just confined to extreme scale machines. Businesses are increasingly using larger scale cluster machines to process the enormous volumes of data they capture about their customers, the goal being to use the data to improve their products and services. Often, such data mining or exploration must be done in real-time, as customers interact with web portals or use search engines. Another class of applications are codes processing sensor data, where it is important to extract information in real-time from data as it is being generated.

This data deluge has created a visible I/O bottleneck. With the increasing size of data, new and innovative approaches to storage are required in order to realize the low I/O overheads users expect. Parallel file systems like GPFS[66], Lustre[18], PVFS2 [17, 46], PanFS [55] provide scalable parallel file system solutions for storage. Similarly, on the enterprise side, vendors like IBM, NetApp, and EMC provide highly concurrent storage hardware and solutions to continue to increase the aggregate throughput offered to ever more data intensive end user codes. However, the limitations of these approaches become evident as systems continue to scale, to hundreds of thousands of cores in high end machines and/or to tens of thousands of nodes in large-scale datacenter systems. For HPC, moreover, a particular issue is the potentially bursty nature of I/O, when hundreds of thousands of cores concurrently write output data. Here, semantic properties of files such as consistent writes, consistent meta data, etc. can limit system scalabilit. Efforts to cope with these limitations have been studied in the high performance domain with LWFS [58] but also within the wide area storage domain with OceanStore [43]. Similar problems have been observed for the metadata services used in distributed file systems, where server contention leads to partitioning and replication (of metadata and metadata services).

New technologies may alleviate but do not solve the scale issues described above. Solid state drives [5] and NVRAM [32] will likely extend the memory hierarchy seen in next generation server systems, causing additional challenges for systems and application software

to efficiently use ever deeper memory hierarchies. Given cost and capacity constraints, however, they may narrow the gap between CPU, memory, and storage speeds, but will not close it, nor will they entirely replace current disk-based storage systems. Instead, they will provide opportunities for new software methods to cache or temporarily store select data, in order to improve application or system performance [8, 56, 68].

The technological solutions described above seek to make additional storage bandwidth available to applications, thereby preventing such codes from blocking on storage operations and reduce the impact of storage operations on code performance. A more fundamental problem is the cost of extracting information from such large blocks of generated or raw data, as it is this cost that determines the end to end performance seen by data mining applications and by scientific end users trying to understand simulation outputs as part of the scientific processes being undertaken. Extracting useful information from data and exploring data to determine certain properties, all such actions require data to be repeatedly written and read to/from storage, as routinely done both in scientific and in enterprise settings [51, 21]. This not only puts yet additional stresses on storage systems, but it also implies further delays and potential bottlenecks in generating scientific or business insights from the increasingly large data sets generated by applications.

Alleviating this problem requires a significant shift in how data management and I/O are performed for high end xmachines. This thesis describes and explores such a shift, which takes advantage of properties of next-generation HPC codes that are not yet exploited by current HPC I/O systems. In particular, we leverage the *latent asynchrony* in these codes and their tolerance for decoupling ancillary I/O operations (e.g., data transformation, formatting for output, non-critical metadata operations etc.) from the compute node's synchronous critical path. Further, by carefully separating and relocating these decoupled operations in space and time from the 'core' HPC code execution, we shorten the application's I/O phases. Finally, we offer flexibility in 'where' and 'when' such operations are performed on I/O data, including:

- **fully coupled:** synchronously and on the source node, at potentially high cost to machines' compute nodes, but avoiding unnecessary data movement or copying;
- **decoupled in space:** *in-band* with the application's I/O actions, but on other nodes, such as on additional nodes on the HEC machine where memory for buffering is less precious and where there are additional CPU resources for operating on data 'in transit' or as it is being moved; or
- **decoupled in time:** *out-of-band*, after data has been moved away from the HPC application, thereby permitting it and the application to proceed independently.

This thesis introduces the data services abstraction, a foundation for the development of technologies that can address increasing data throughput requirements, while allowing for lower time to extract information from the data set. A data service can be defined as a pipeline of coupled components operating on the output data to both extract information and to add metadata which can further aid the analysis of the scientific data. The components of data services can be full fledged analysis codes such as pairwise bond computation codes used in molecular dynamics or smaller utility based operators such as data reduction and aggregation operators. A key factor that enables the development and deployment of data services is introduced in this thesis, namely data staging. Data staging is a set of resources allocated "near" the application, usually offering a high throughput, low latency interconnection to the application nodes. Complementary to staging is the idea of managed data movement to alleviate the performance impact on the application from the movement of data by combining server directed I/O with scheduling techniques to reduce interference. Finally, this thesis goes beyond simply allocating discrete resources for staging but introduces the concept of staging on the application nodes or JITStaging. Using dynamic code generation, the staging area is extended to utilize the application memory and computational resources to open up more functionality for the data service without additional operational costs within the staging area.

1.2 Thesis statement

In transit services for data extraction, management, and processing provide a high performance path for modern scientific application and workflows. For such ”data services”; the use of staging coupled with online processing of data creates an inherently scalable approach to dealing with the increasing mismatch between processing and storage. Service instances go beyond just simple data store/forward to open up complex processing actions that enable improved time to solution over traditional approaches performed after data has been stored. Coupled with novel approaches such as zero overhead transport protocols like RDMA, the use of structured data serialization, data transport scheduling, and intelligent file formats, data services can provide significant additional data processing functionality while also improving application performance.

1.3 Contributions

The contributions of this thesis are two fold. Firstly, a new abstraction is introduced, Data Services, which enables timely data processing and insitu computation and visualization as data sizes and data rates scale in the current and next generation of scientific applications. Secondly, this thesis describes techniques and mechanisms for the efficient movement and processing of the data generated by large scale data intensive simulations.. These techniques include the use of one sided communication such as RDMA, which allows for low overhead movement of data without local processing impact; and server side I/O scheduling, which enables the management of asynchronous movement to allow for low overhead data reordering as well as contention avoidance to reduce the impact of resource contention on the application. Finally the use of dynamic code generation coupled with easy ”code” movement in the data pipeline forms the foundation of future research to optimize the placement of data manipulation functionality to fulfill quality of service (QoS) requirements imposed by the user.

1.4 Impact of future technologies

This thesis provides a software, and more specifically a middleware, solution to the data management problems as we approach tens of petaflops and gear up towards exascale. Hardware design solutions have also been proposed in order to achieve similar goals. New technologies such as Solid State Disks (SSDs) and Non-volatile Memory (NVRAMS) are seeking to address the challenges to exascale data also, however the approach is orthogonal and complementary to the approach presented in this thesis. In particular the staging technique presented here can be utilized even more efficiently when combined with fast persistent storage hardware. This is discussed in more detail in Chapter 7.

1.5 Thesis organization

The thesis is organized as follows. In Chapter 2 we layout the background behind the data service abstraction, including some of the key applications which have driven this vision. Additionally we describe the past technological artifacts that have served as the foundational components of the design and implementation of data services. The data service abstraction and the key points of their design as well as the implementation goals are described in Chapter 3. Following chapters describe in detail the key innovations necessary for the successful development and adoption of data services. In Chapter 4 we introduce the *staging area* as well as the key step in realizing low impact asynchronous data movement to the staging area, viz. the perturbation avoidance scheduler. The standard staging area is further elaborated on with dynamic code generation and code movement in Chapter 5 which utilizes not just the computational capability of the staging area but also the processing cores in the application to optimize data operations. Here we show how using the knowledge about the global data view which is available in the staging area and specializing data operations in the application area can be used to create novel data services with minimal performance impact on the application runtime. Finally we show the utility of data services in Chapter 6 in conjunction with two of our key leadership applications,

CHIMERA and GTC. In Chapter 7 we look at the future directions for research that are opened up based on the initial contributions of this thesis and we present our conclusions in Chapter 8.

CHAPTER II

MOTIVATION AND FOUNDATIONS

This chapter describes the motivation and the background, which has driven the design and development of the data service abstraction. Data services provides a new paradigm for outputting, storing and processing scientific data from 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 unique I/O challenges raised by each. The data service abstraction can be considered to be one way of addressing these challenges.

2.1 Motivating applications

Our research is largely driven by the experiences of computational scientists dealing with the problem of data management on leadership machines. Concrete sample applications have been chosen to drive the motivation and design parameters, as well as providing us with an evaluation platform for our research efforts.

2.1.1 WARP

WARP (and its successor LAMMPS) is a molecular dynamics application utilized by collaborators at Georgia Tech conducting research on material deformation and crack propogation. Molecular dynamics codes aim to replicate atomic-scale motions, using various representations of the forces involved. These codes are used to study some very large problems, sometimes involving hundreds of thousands to millions of atoms. The run-time of such problems can be quite long, even on massively parallel machines, and therefore the task of visualizing and steering of the codes can be important. Traditional methods of dealing with such data flows involve complete state logging for later viewing (which does

not scale well to large sizes or long runs) or the storage of partially interpreted data such as auto-correlation functions or time averages (which may fail to preserve data needed in subsequent interpretation).

2.1.1.1 Description of I/O challenges

We use WARP as an example of a representative application where the output data is processed through an extensive pipeline to extract information from the data. In the case of WARP, this pipeline consumes the molecular data produced by the parallel application in the *Bonds* analysis application. Bonds performs a pairwise computation on all of the particles in the system to calculate a set of "bonds", two molecules that are within a threshold $R_2 < \epsilon < R_1$. Due to the complexity of the computation, the Bonds application can have a significant slowdown effect on the WARP application's execution. The WARP molecular output data is also stored on disk, either before the Bonds step is performed or after.

2.1.2 GTC

GTC, the Gyrokenetic Toroidol Code, is a particle-in-cell simulation of plasma as part of the fusion process. GTC has very stringent scalability and I/O requirements, running on more than 150k+ cores on Jaguar XT5 at ORNL. On current generation machines, GTC generates 100 terrabytes of data per day for analysis [39] with plans for an order magnitude increase on next generation machines. Due to these requirements, GTC serves as an ideal platform for the development of new mechanisms for I/O and data management.

2.1.2.1 Description of I/O challenges

GTC is a representative example of a highly scalable application producing data at next generation data rates. We study both the GTC particle output as well as the restart output, both outputs fulfilling the requirement of being extremely large output data sets that can produce a significant impact on application performance. In our experiments, we configured GTC to output 180 MB/process, and utilized weak scaling to test outputs ranging from

22.4 GB to 360 GB for each output timestep. We also tested the scaling performance for GTC

2.1.3 CHIMERA

CHIMERA[52] is a multi-dimensional radiation hydrodynamics code designed to study core-collapse supernovae. We look at the periodic restart output that is used for both checkpointing and post-processing. The restart data consists of 80 scalar and arrays. Global arrays are regularly distributed among a 2-D grid of MPI processes. CHIMERA uses the ADIOS API [50] for I/O allowing multiple methods to be compared by simply modifying a variable in the configuration file. The data is defined as part of an external XML configuration with both structure and meta information and enabling the use of structured FFS data for output purposes. We have instrumented the application with specific calls to ADIOS in order to provide phase information to the underlying transport method, allowing us to customize the behavior of the data transport.

2.1.3.1 Description of I/O challenges

While the CHIMERA data output is not pushing the boundaries in terms of data volume, a unique aspect of CHIMERA is the requirement for the data output format to be HDF5 [64]. In our experiments, we evaluate the service to convert the output data to HDF5 format within the staging area and compare it to the native output method.

2.2 Foundational technologies

While data services are a significant change in the paradigm of data management for high performance applications, their development has been focused through the use of existing foundational technologies. Three components in particular have played a significant role in the motivation for the data service abstraction.

2.2.1 FFS

The *Fast Flexible Serialization (FFS)* library is on the primary component of the data service stack. FFS is a descendant of the *Portable Binary IO (PBIO)* [27, 15] that provides an interface to serialize data into a tightly packed self describing message. FFS uses descriptive data formats to allow remote consumers to query the data format and extract structural metadata from the message. In this regard, FFS is similar to an XML message that can be used for data discovery without a priori knowledge of the data formats. While FFS does not produce data streams in human readable formats, FFS does provide several distinct advantages over the XML messages such as the compact representation of FFS, ability to be converted into XML messages and the fast marshalling and unmarshalling of data. These capabilities make FFS particularly suited for use the HPC environment, and we have utilized FFS as the messaging format for our implementation of the data services abstraction.

2.2.2 EVPath

EVPath is an event delivery middleware developed as a replacement for the publish-subscribe ECho middleware [26, 28]. A key property of the EVPath middleware is the idea of separating the control plane which handles the creation of the overlay network of ”stones”, from the data plane which deals with the efficient movement of data within the network of stones. EVPath has been extensively used in research on monitoring [45], data aggregation [44],

2.2.3 ADIOS

The ADatpable I/O System (ADIOS) is a componentized I/O framework developed jointly at Georgia Tech and Oak Ridge National Laboratory [48]. ADIOS provides a framework for production applications to utilize both established I/O methods as well as experiment with new research technologies such as those described in this thesis. Implementing the data service abstraction within the framework of ADIOS, provides the technology with a

large number of adopters in the scientific computing domain. In particular, the nuclear fusion particle-in-cell simulation, GTC has been an early adopter of the data service abstraction in order to deal with problems arising from the data output requirements of the applications.

2.2.4 Evaluation platforms

This thesis utilizes the problems faced by these real applications and builds on the foundations of the aforementioned technologies to describe the data service abstraction, described in detail in the next chapter, has been implemented on three computing platforms; viz. the Cray XT4/5 Jaguar, a leadership class supercomputer at Oak Ridge National Laboratory, and on the Georgia Tech CERCS Rohan cluster, a SDR infiniband based linux cluster of 50 nodes with two cores each and the Georgia Tech CERCS Maquis cluster, a QDR infiniband based linux cluster of 16 nodes with 8 cores each.

The motivational applications and the foundational technologies enumerated in this chapter have provided us with the challenges faced by scientific applications today. In the following chapters we look at the design of data services, the key implementation areas and how data services, when used with these applications, can overcome these obstacles.

CHAPTER III

THE DATA SERVICE ABSTRACTION

Current methods for dealing with the previously mentioned data deluge, such as increasing storage bandwidth, do not scale to petascale and beyond for both cost and performance metrics. The conventional practice of storing data on disk, moving it off-site, reading it into a workflow, and analyzing it to produce scientific solutions becomes harder due to large data volumes and limited backend speeds. In fact, even the time required to move data to storage can become an obstacle, since if output actions cause an application to block on I/O, countless numbers of compute cores sit idle waiting on output. In addition, science is affected if it takes say, 500 seconds to output data, since that makes it hard to justify writing data more than once per hour, even if the science may benefit from more frequent output.

Instead, we propose a new abstraction for data management that can satisfy the requirements for today's applications as well provide a scalability path for the next generation of leadership systems and applications. The Data Service abstraction merges the data output stage in I/O with the data management process required for performing both pre-analysis and analysis on the data. Our approach goes beyond simply accelerating output to also moving select data manipulation tasks traditionally placed in an offline workflow 'into' the fast data path on the supercomputer. Suitable tasks include those that reduce data without loss of scientific validity, generate metadata (e.g., indexing) for easier data access, or perform lightweight analysis tasks for online validation of application 'health' via dashboards. While previous work has demonstrated the utility of this approach [73, 1, 10, 31], what remains lacking are system abstractions and the underlying runtime support for efficient I/O task representation, deployment, execution, and management.

Data Service is a system-level abstraction that encapsulates methods for data movement and manipulation on the fast path of data output. Data Services are created and deployed separately from application codes, thereby separating changes made to application codes by science users from changes made to I/O actions by developers or administrators. Data services can be run asynchronously from the large-scale simulations running on supercomputers, in order to decouple I/O actions and their performance behavior from the those of the computations performed by large-scale applications. Thus, science end users can focus on their codes, and administrators or developers can help create and manage I/O processes.

To make I/O processes manageable, data services are associated with I/O so as to retain flexibility (1) in the resources they consume, (2) in where services are run (e.g., on compute nodes and/or on staging nodes), and (3) in how and when they are run, including explicitly scheduling their execution to avoid perturbing the petascale simulation [3]. Flexibility in the levels of resources dedicated to I/O ranges from 'none', where compute nodes are used to run output actions, to cases in which a substantial number of 'fat nodes' are used for staging data and manipulating it prior to storing it on disk. Flexibility in service placement includes placing certain data reduction actions close to the source, even directly on compute nodes via the *JITStager* abstraction used for capturing output data. Data produced by JITStager can be fed to storage and/or to additional data services placed on staging nodes, where data may be buffered, annotated, or reorganized prior to moving it to storage. Flexibility in how services are run is supported by output scheduling methods that take into account supercomputer interconnects with respect to perturbations of simulations caused by output activities, and by backend-sensitive methods that adapt to different storage characteristics and behaviors. Examples of the latter are those that avoid using storage targets already used by other parties or more simply, that use the appropriate number of storage targets to maximize throughput.

The data service abstraction makes it easier to develop, deploy, and maintain per-site and per-run optimizations of I/O processes. The abstraction also exploits the facts that there

exist many non-trivial I/O and data processing tasks that can be done with few additional computational resources, on compute and/or on staging nodes, and moreover, that performing such tasks can result in performance gains when writing data and/or in usability gains by increasing the information content of data through annotation. Toward these ends, data services are defined to differentiate between (1) data extraction, using the JITStager abstraction, (2) data processing via light-weight computations associated with data services, and (3) data storage using methods that take into account storage system characteristics and behaviors, and (4) for flexibility, the implementation of data services separates data movement and manipulation – the data plane – from how such actions are managed – the control plane. Therefore, new scheduling methods for data extraction or new techniques for how storage targets (or other backends) are used can be deployed easily, without changing data plane movement and processing actions.

Data services have built on the ideas proposed in previous work on Service Augmentation [73] and the LIVE Data Workspace [1]. The data service itself can be considered to be comprised of three distinct parts, the data input, the data manipulation operation, and the output. By capturing the discrete and separate nature of the data input and output processes, we can describe data services that bridge between different network architectures as well as providing the basis for services, which receive data from the network and output data to storage. The data manipulation operator is a coherent component that can be merged with differing I/O components allowing for maximum flexibility in the construction of the final data pipeline.

More formally, a data service can be described as follows

$$A \overset{input}{\rightarrow} f(A) \underset{output}{\rightarrow} A'$$

where A is the input to the data service and $A^{'}$ is the output from the service. In addition, we can describe the data pipelines as a combination of the services such that the input for service, S_i is the output from a service S_{i-1}. Defining an entire data pipeline as a series of connected services carries a distinct advantage towards future efforts to create

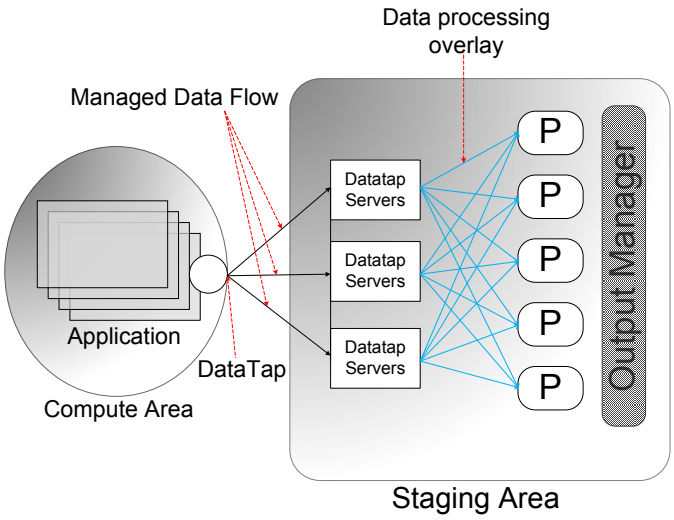


Figure 1: High level view of the Data Service Architecture.

execution models and utilize service placement optimizations.

3.0.5 Architecture and design

A *data service* is a collection of actions on 'in transit' data, carrying out tasks like data extraction, data staging, formatting, indexing, compression, and storage. A high level depiction of the Data Service architecture appears in Figure 9, which shows that conceptually, running a Data Service has the following phases:

- from bytes to structured data: writing and formatting output into buffers to create data items of well-defined structure and memory layout;
- controlled data movement: extracting data items to maximize application performance;
- online data processing: applying service codes to output data while it is being moved.

In order to support customizable and configurable I/O data movement and processing, each of these steps are carried out in ways that are defined by control methods associated with them. Examples include the scheduling of data eviction from compute nodes and the controlled movement of data to storage subsystems. Each of these steps and their controls are explained next.

3.0.6 Structured output data

With the massive quantities of data generated by petascale applications, it is not uncommon that only a fraction of this data is actually required for scientific analysis. Thus, producing and outputting the entire data set and then later extracting a smaller portion for post-processing can create an unnecessary performance bottleneck. However, identifying "useful" data is highly specific to each scientific undertaking, requiring in-depth knowledge from the user as well as application hints that enable this reduction. For example, for a molecular dynamics application, a user may only be interested in the characteristics of particles in a small bounding box. Traditionally, this requires the application to provide the functionality that allows output in a bounded space, thus making it necessary for end users to change application code. In contrast, data services allow end users to flexibly associate such functionality with the data output process itself. Getting access to then utilizing this meta information is very important for the data service. In addition to creating structured output, the data service must also ensure that the overhead associated with storing the metadata is small enough to not hamstring the users.

Alongside the structured self-describing output it is extremely useful for data services to add data specific annotations that can further facilitate downstream analysis and knowledge extraction. With massive data sets being generated from an application at high frequency, it is often impractical for the analysis codes to go through the incoming data with a fine toothed comb. By using data annotations that are added at the source of data generation this process can be greatly simplified. Useful examples of annotations include data characteristics such as min/max, statistical properties, bitmap indices, and probability distribution functions amongst others.

3.0.7 Controlled data movement and the staging area

Once the data has been generated by the application, the next step is to move the data to available compute cores for processing or to the disk for storage. For both these scenarios,

the Data Service has to provide a mechanism for moving the data off node. In order to lower the overhead associated with data movement, we sought to investigate and develop optimized data movement technologies such as RDMA-utilizing asynchronous I/O, while also using the unique capabilities of data services for managing the transfers to minimize application impact.

One concern for creating the data processing pipeline has been variability in the storage performance of modern scalable disk systems. Due to the variation in the number of users using a shared machines such as Jaguar at Oak Ridge National Laboratory, a mechanism is needed to insulate the applications from usage spikes and consequent delays experienced for their I/O actions. In order to achieve this, we propose the concept of a staging area. The staging area is an intrinsic companion to the data service model. In short, the staging area is a collection of resources close to both the application nodes and the output ports serving as a transit point for data movement. Instead of directly moving data to disk, data services operate by moving data first to the staging area. Once the data is available on the staging area, the service can utilize the available compute capability for processing the data in order to fulfill the service requirement. Once the data has been serviced, it is pushed out to storage. Intermittent performance problems with the storage area do not cause application slow down using this method, because the staging area provides a large buffer, similar to how an inductor works in an electronic circuit.

Moving data to the staging area also yields higher throughput for data movement because the limitation of disk bandwidths are avoided as long as there is available memory for buffering. By utilizing asynchronous data movement, the throughput bottleneck is made insignificant by allowing the application to proceed with computation as data transfers are initiated. However, this asynchronous data movement can be a cause of jitter within the application, especially on scalable network architectures like the SeaStar/SeaStar2/SeaStar2+ on the Cray XT3/4/5 where data movement can directly interfere with intra-application communication. In fact, even the small levels of jitter in the application communication

can result in large performance penalties as the application scales up to 100,000+ computational cores. Addressing the challenges posed by this limitation, has led us to develop new methods for data movement scheduling that avoid interference by carefully selecting the time windows when data movement occurs in, or by carefully limiting the rate at which data is streamed out of the node reducing the likelihood of interference.

3.0.8 Online data processing

As data moves from the generation point to the eventual data consumer (e.g., to disk storage or to an online data visualization), there is both a necessity and an opportunity for in-transit processing. Necessary processing services include those that convert data into the standard forms required by the backend, such as the HDF-5 or NetCDF formats used in file systems. Performing data formatting in data services 'offloaded' to the staging area can help reduce these overheads. Other examples of online data processing operations are those that seize opportunities for performance improvements through data reduction, improve the accessibility of output data through data indexing, or perform tasks meaningful to applications like generating histograms or validating output data.

The Data Service architecture shown in Figure 9 identifies two points where online processing can occur. The availability of computational capability within the staging area allows the service to be scheduled on the staging area. Due to the nature of the staging area, it also serves as a point of aggregation within the pipeline - many application nodes transfer data to a single staging node. This aggregation characteristic can be exploited by the service to perform operations that would, on the application node, require inter-node communication without the necessity of this communication step. Even global operations can be scaled better due to the reduction in the size of the staging area compared to the application.

While the staging area provides a natural location for service execution, it is limited by the reduced computational capability compared to the application cohort. Thus in terms of

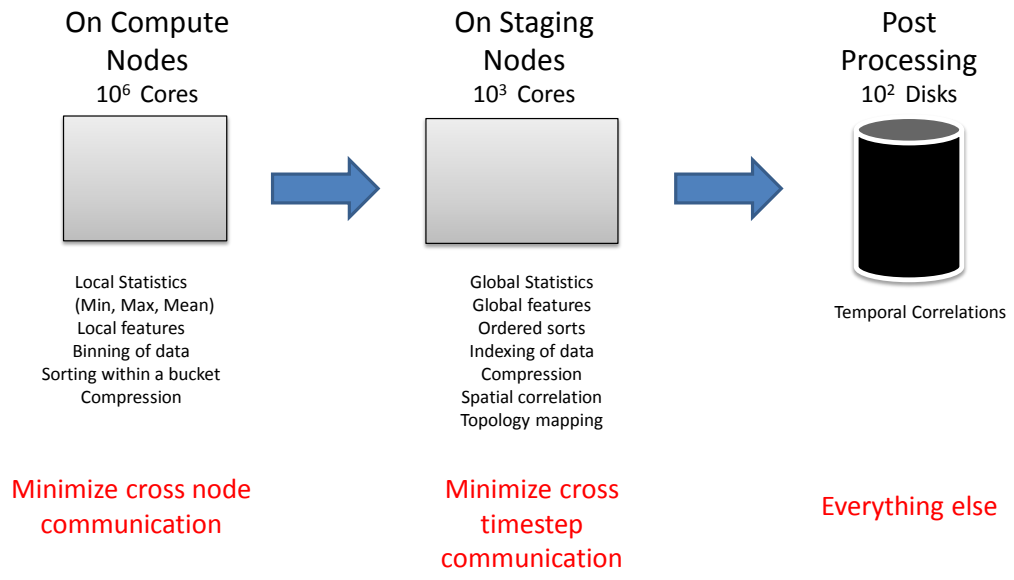


Figure 2: Heuristics to select where code execution takes place in the I/O pipeline.

raw FLOPS the staging area provides only a small fraction of the capability of the application nodes. By utilizing the application nodes for partial service processing we can gain the benefits of both available platforms. Operations executed within the application node must be restricted to independent data operations that do not require cross node communication. Annotation operations where the output data is tagged with characteristics used in downstream processing, or data manipulation operations such as those which filter or reorder the data, can benefit greatly by being executed within the application address space[4].

A final family of operations able to benefit from the data services is post processing analysis. The availability of multiple timesteps, and even multiple application runs, allows for a wider range of temporal analysis to be performed. Although we do not consider this set of operations in this thesis, the upstream data preparatory functions can significantly enhance the performance of post processing analysis [75]. Figure 2 shows the basic heuristics we can use to select the part of the data pipeline to execute different operations in.

The utility of data services hinges greatly on the availability of a data transport that can provide low overhead, low impact data movement. As mentioned in this chapter, the

transport must provide a usable asynchronous data movement mechanism using RDMA (or RDMA-like one-sided APIs) in order to minimize the time spent by the application in the data movement operation. The next chapter provides a detailed description of the design of a data transport the fulfills these criteria.

CHAPTER IV

ASYNCHRONOUS DATA MOVEMENT AND SCHEDULING

Asynchronous methods are known to help in addressing the I/O needs of high performance applications. In [12], for instance, the authors show that when asynchronous capabilities are available, synchronous I/O can be outperformed by up to a factor of 2. The studies performed in this thesis use a novel, high performance data transport layer developed by our group, termed DataStager. DataStager is comprised of a library called DataTap and a parallel staging service called DataStager. In order to support easy inclusion of best practice, scalable I/O in high performance codes, others within our research collaboration have implemented the ADIOS I/O portability layer [48], which supports both blocking and asynchronous I/O modules. DataTap interfaces with the ADIOS API in order to keep application level code changes to a minimum and to enable the user to determine the transport of choice at runtime.

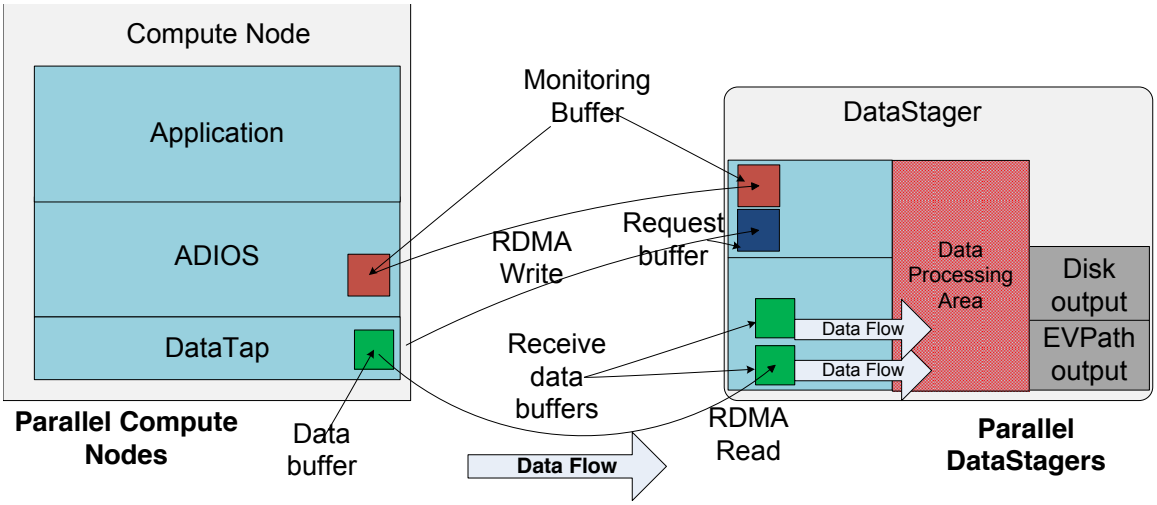


Figure 3: *DataStager Architecture.*

One mechanism that has been used to manage asynchronous communication is server-directed I/O[58, 59]. This is particularly useful in high performance architectures where a small partition of *I/O nodes* service a large number of compute nodes. The disparity in the sizes of the partitions, coupled with the bursty behavior of most scientific application I/O [54], can lead to resource exhaustion on the I/O nodes. In server-directed I/O, the data transfers and hence the resources, are controlled by the I/O nodes, allowing smoother accesses. Such techniques have been used for both large cluster filesystems [58] and for disk-directed I/O [42]. When asynchronous communication in an RDMA environment is added, server control becomes doubly important. Specifically, in addition to managing the resources, the server control of the data transfer allows the application to progress without actively *pushing* the data out. Instead, the server *pulls* the data whenever sufficient resources are available. Under ideal conditions, the rate at which the server pulls the data – the ingress throughput – is equal to the rate at which the server retires the data – the egress throughput. The ability of the server to satisfy bursty ingress requests will naturally be bound by the interconnect bandwidth between the I/O node and the compute partition.

We address the problems of scaling application I/O to petascale and of the need for runtime understanding and analysis of data by utilizing managed asynchronous data movement. The asynchronous operation reduces the impact on the application from ”blocking”, i.e. the time spent waiting for the completion of the data transfer. The potential jitter introduced is minimized by managing the timing for the data movement operations, carefully scheduling them to not overlap with collective communication requests issued by the application.

The DataStager-DataTap architecture integrates into data services by utilizing the self-describing binary format, FFS [15, 29]. This makes it possible for binary data to be inspected and modified in transit [72], and it enables the association of graph-structured data processing overlays, termed I/OGraphs [1]. With such overlays, I/O can be customized for a rich set of backend uses, including online data visualization, data storage, and transfer

to remote sites, including with standard methods like GridFTP [16]. DataTap is also integrated with the ADIOS interface as a custom data output method. This both encourages adoption and provides an easy to use interface to DataTap for a large variety of applications.

The DataStager-DataTap system was initially developed on the Cray XT class of machines using the Portals programming interface [14, 13]. We have also implemented a version using the Infiniband uverbs interface; performance evaluation of the infiniband version is included in [1]. It is noteworthy to mention that like all asynchronous I/O efforts, the DataStager can only service applications that have sufficient local memory space to buffer the output data.

4.1 Previous work

There has been significant prior research into studying improvements to the I/O performance for scientific applications. Highly scalable file systems like PVFS [17, 46], Lustre [18], and GPFS [66] are examples of efforts to improve I/O performance for a wide range of applications. Although file systems such as GPFS do offer asynchronous primitives, there has been no effort to study and eliminate interference of asynchronous I/O and intra-application communications in these file systems. LWFS [58, 59] is a high performance lightweight file system designed to eliminate metadata bottlenecks from traditional cluster file systems. The current implementation of LWFS is very close to that of the DataStager, offering an asynchronous RPC and a server-directed data transfer protocol. The scheduling mechanisms described in this chapter are orthogonal to the functionality of LWFS and can be used in order to further improve application performance.

Recently, there has been an effort to consider data staging in the compute partition in order to improve performance. [57] is an effort to improve I/O performance by using the additional nodes as a global cache. Since I/O delegates are implemented as part of MPI-IO the advantage of this approach is generality. However, the performance impact of this approach is limited for large data outputs where the I/O delegates exhaust the available

caching space.

PDIO [69] and DART [23] provide a bridge between the compute partition and a WAN. Similar in design to our data staging services, both platforms would potentially suffer from similar problems with interference. The idea behind managed data transfers in DataStager could be utilized by both projects to reduce the impact of asynchronous I/O on application performance. As part of our future work, we are also addressing the connection to WANs through the EVPath [26] messaging middleware.

Hot spot detection and avoidance in packet switched interconnects [37, 34] and in shared memory multiprocessors [20, 11] are related to our efforts to reduce interference with communication in scientific application. Solutions to the problems in those domains are still significant on the highly scalable MPP hardware, we are targeting, where state-aware scheduling provides a software-only solution to the problem of contention.

In [12] the authors evaluate MPI non-blocking I/O performance on several leading HEC platforms. They found only two machines actually support non-blocking I/O and benchmark results showed substantial benefit by overlapping I/O and computation. The characteristics of the benchmark used, however, does not allow the authors to study the impact of the overlap of I/O and computation for asynchronous I/O. In our work we have discovered that the real performance penalties for asynchronous I/O are from interference with communication.

[61] studies the impact of different overlapping strategies for MPI-IO. The authors consider different strategies for the overlap of I/O with computation and communication showing the performance benefits of asynchronous I/O. However, the results are limited to a small number of processors, and as we show, there is limited interference at these sizes. The innovative benchmarking tool used can be an aid to our own effort in developing better strategies for data extraction.

Overlapping I/O with application processing has been shown to dramatically improve performance. SEMPLAR [7, 9], built on the SDSC Storage Resource Broker, supports

asynchronous primitives allowing asynchronous remote I/O in the Grid environment. Because SEMPLAR uses a separate thread to implement a push-based model, there is a smaller likelihood of interference with application communication. However, the authors observed a performance decrease in some scenarios where resource contention penalized performance. Due to their push based model, the solution involved a reorganization of the application code to remove overlap between I/O and MPI. The DataStager provides a server-based mechanism for accomplishing the same task, while keeping application modifications to a minimum.

4.2 Design

DataStager has two different elements: the 'DataTap' client library and the 'DataStager' processes. The DataTap client library is co-located with the compute application. It provides the basic methods required for creating data objects and transporting them, and it may be integrated into higher level I/O interfaces such as the new ADIOS interface used by an increasing number of HPC codes[48]. The DataStager processes are additional compute nodes that provide the data staging service to the application. The actual data output to disk, data aggregation, etc. are performed by the DataStagers. The combined libraries work as a request-read service, allowing the DataStagers to control the scheduling of actual data transfers.

Figure 3 describes the DataStager architecture. Upon application request, the compute node marks up the data in FFS format (see [27] for a detailed description of PBIO, an earlier version of FFS) and issues a request for a data transfer to the server. The server queues the request until sufficient receive buffer space is available. The major costs associated with setting up the transfer are those of allocating the data buffer and copying the data; they are small enough to have little impact on overall application runtime. When the server has sufficient buffer space, an RDMA read request is issued to the client to read the remote data into a local buffer. This data is queued in the DataStager for processing or directly for

output.

4.3 Managing data transfers

DataStager's scheduling service, implements server directed I/O. Use of server-side I/O allows us to explore novel methods of scheduling data transfers as part of the service. We have designed four schedulers in order to evaluate their ability to enhance functionality, to provide improved performance, and to reduce perturbation for the application:

1. a constant drain scheduler,
2. a state-aware congestion avoidance scheduler,
3. an attribute-aware in-order scheduler, and
4. a rate limiting scheduler.

```
Node(R) is the originating node for request R;
Size(R) is the size of the I/O request;
if Node(R) is waiting for completion then
    return TRUE;
else
    t^n_start, t^n_end are the start and end time for compute phase n;
    t_current is the current time;
    t_iter is the estimated width of a single iteration for Node(R);
    t_request is the time at which the request was issued;
    Δt = t_current - t_request ;
    Δiter = floor(Δt / t_iter) ;
    foreach compute phase, i in Node(R) do
        if t_current is between t^i_start AND t^i_end then
            return TRUE;
        end
    end
    return FALSE;
end
```

Algorithm 1: The Phase aware scheduler determines whether the application is in the compute phase for the DataStager to start the transfer.

DataStager uses resource aware schedulers to select requests for the RDMA service. Selection of a request from the transfer queue is based on the following discrete steps.

- **Memory check.** A check is performed to determine whether there is sufficient buffer space available to service the request. This check ensures that the DataStager does not issue unbuffered reads and suffer from resource exhaustion.

- **Waiting check.** A node may issue an asynchronous I/O request and then block for completion after a period of computation. If a node is currently blocked waiting for a request to complete, the request should be serviced as soon as possible to minimize the performance penalty.

- **Scheduler Check.** A schedule request is made to the scheduling module for each transfer request. The transfer is only initiated if all the scheduler modules indicate viability. This enables the DataStager to *stack* schedulers in order to fulfill multiple resource allocation policies while also simplifying the development of new schedulers.

Once all schedulers have agreed to issue a transfer request, it is serviced in two parts. First, an RDMA read request is issued to the originating node. Due to the latency of request completion and because available buffer space is usually larger than a single request size, multiple requests may be serviced simultaneously. This overlapping enables DataStager to complete all requests faster. Once the RDMA read is completed, an upcall is made to the staging handler. The handler will then process the message according to the configured policy – direct write to disk, network forwarding, further processing, and so on. The incoming data is in FFS format allowing the use of FFS's reflection interface to query the data block and perform operations such as aggregation and filtering in the data processing area without making a copy. The data can also be published with the EVPath [26] event transport for further processing as part of an application specific I/O pipeline [72].

4.3.1 Continuous drain scheduler

The Continuous Drain (CD) scheduler is designed to provide maximal usage of the buffering available to the staging area. As soon as buffer space is available, an RDMA read call is

issued. The throughput for this scheduler is limited by the ingress and egress throughputs, the rate at which the staging area processes data and the amount of buffer space available in the DataStager. However, it also creates a large impact on the performance of the application (and hence also secondary effects on the ingress throughput). For large cohort sizes (i.e., a large number of clients), the strategy of draining the data as fast as possible can substantially perturb the time taken to complete intra-application communication, particularly large collective calls like MPI_ALLTOALL. In fact, the resulting overhead has an impact on performance that dwarfs the time spent waiting for synchronous data transfers to complete. Interestingly, despite that level of perturbation, the CD scheduler can yield good performance in cases where an application does not rely on collective global communication or uses asynchronous MPI communication.

4.3.2 Phase-aware congestion avoidance scheduler

As stated earlier, the use of asynchronous methods for data transfer can reduce or eliminate the blocking time experienced by HPC codes using synchronous methods for I/O. A resulting new problem is one of potential perturbation in communication times experienced by tightly coupled and finely tuned parallel application. This is because of the overlap of intra-application communication (e.g., MPI collectives) with the background transfer of output data that uses the same interconnect. Interestingly, this phenomenon is not generally observed for smaller scale parallel codes (e.g., up to a hundred nodes), but as the application scales to larger machines such as the Cray XTs (i.e., to multiple hundreds of nodes and above), it can significantly impact the performance of intra-application communications and thus, of the application itself.

The contention caused by multiple nodes using the interconnect can significantly decrease communication performance. Although the increase in perturbation is not surprising, as we scale to more than 512 nodes, we observe that the total perturbation cost is far greater than the that of simply blocking for I/O. Moreover we find that the perturbation

caused by the background transfer of data is not limited to the asynchronous DataTap-DataStager method. As can be seen from Figure 7, the function *smooth1* has an overhead with both POSIX and MPI-IO synchronous methods.

In order to prevent application perturbation, the phase-aware scheduling mechanism attempts to predict when each process is involved in either a local computation (*compute phase*) or in an MPI communication (*communicate phase*). Such phase information is provided to the DataStager through a 'performance buffer' that is maintained for each node communicating with the DataStager (see Figure 3). On the compute nodes, the DataTap library updates its local performance statistics at the end of each iteration. If the client detects a significant change (e.g. the current iteration run time exceeds the previously reported one by more than 10%), the client updates a remote performance buffer on the DataStager.

A key requirement for phase-aware scheduling is to accurately estimate the duration of computational and/or communication phases of parallel codes. Specifically, the scheduler must estimate when the application transitions to a compute only state, where such a state is defined as an application state with which is associated no more than some small amount of communication (i.e., only isolated send/receives). This is because simple point to point and sub-communicator broadcast communications (or pure computation) are not likely to be perturbed by asynchronous I/O as opposed to global MPI collective operations.

One way to estimate the duration of computational application phases is to solicit input from developers, by asking them to mark the portions of the application code that are suitable vs. not suitable for background I/O. ADIOS provides a uniform API for these types of hints. For simplicity, we currently use this approach, but this can be generalized and automated using known methods for phase detection, including the techniques reported in [22].

An application enters a compute state at time t_{start} and computes for a time Δt, exiting the compute state at time t_{end}. In some scientific applications, the period Δt is regular,

i.e., once the application reaches steady state there is very little variation in the time spent in each compute state. For applications where the computational loop is irregular (e.g., optimization applications), a different mechanism needs to be studied for implementing the predictor. For regular applications (such as our motivating application, GTC [60]), a perfect phase-aware scheduler would always start a data transfer after t_{start} and finish the data transfer before t_{end}. Given sufficient iterations between successive I/O calls, such an ideal scheduler would induce no interconnect perturbation and have a minimal performance impact.

In our current implementation, explicitly installed instrumentation is used to inform the I/O library each time the application has entered a computation state, at time t_{start}, and at time t_{end}, the I/O library is informed that the application has left the computation state. Because all of the I/O requests will not be serviced within a single application iteration, the I/O library also tracks the time taken to complete one application iteration (the *main loop*), t_{iter}. The performance tuple for n compute phases in one application iteration, $\{t_{iter}, [\{t_{start}, t_{end}\}]^n\}$, is lazily mirrored on the DataStager, as described previously.

Experimental results attained with this scheduler and shown in Section 4.4 show that the phase aware scheduler can reduce the performance impact of background I/O from more than 10% with POSIX data output to about 2% even as the application scales to more than 1024 nodes.

4.3.3 Attribute-aware in-order scheduler

One disadvantage of using a state-aware scheduler is the burden placed on the data staging buffer space in order to create an ordered stream of output data for those applications that require one. This can result in reduced performance due to the additional time that data blocks are held in buffer instead of being processed and transmitted (or written to disk). One example of where such a problem arises is writing a snapshot to a shared file. To complete the write of block b_i, we need to know the sizes of blocks $b_j | j < i$. Instead of

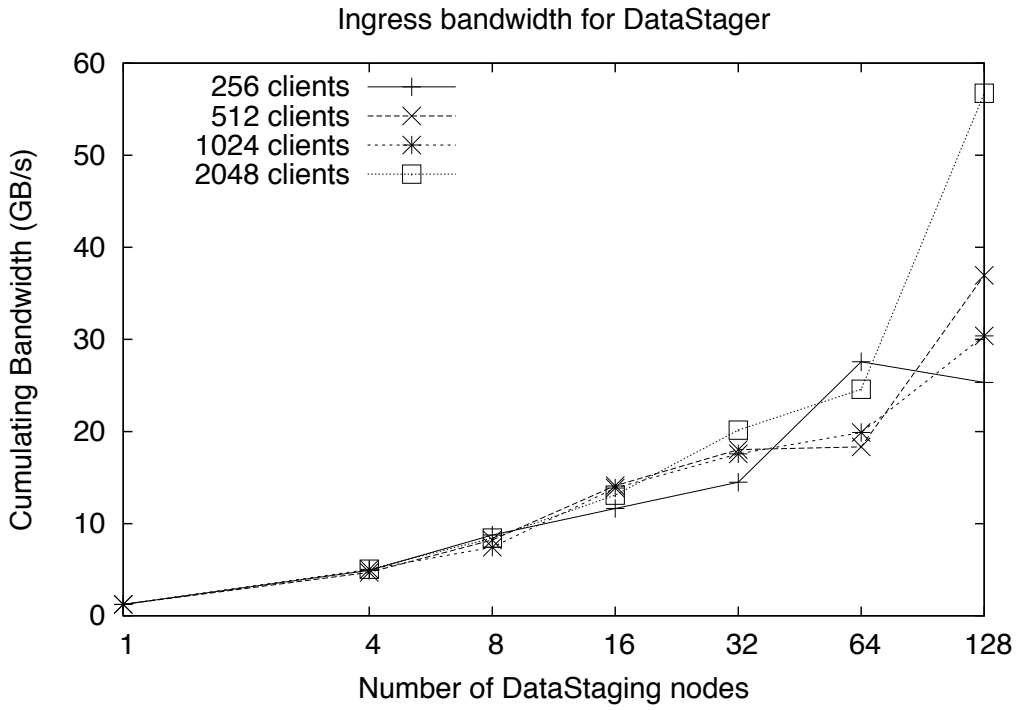
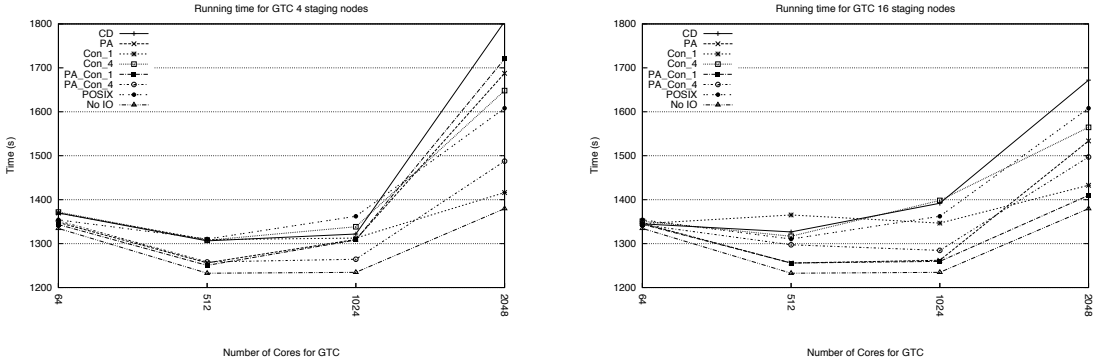


Figure 4: This benchmark compares the cumulative and per DataStager throughput observed when running on 1024 nodes.

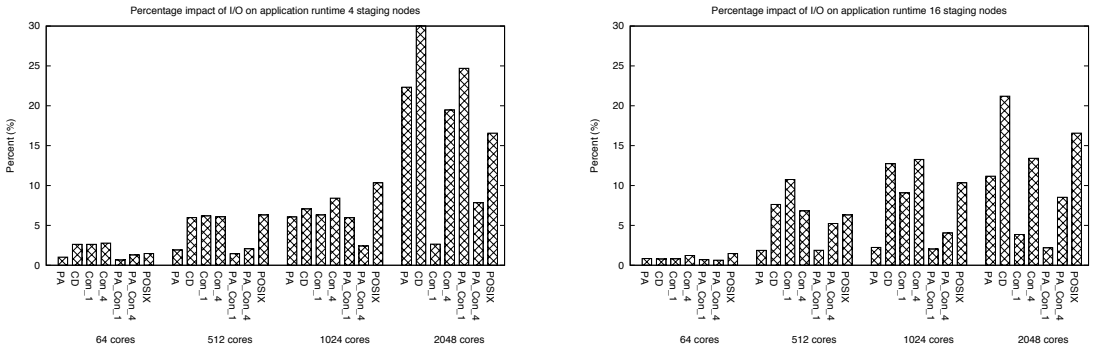
letting the compute application synchronize itself and exchange sizes, we propose that the data staging service can more simply perform this operation with less overhead.

We address this problem with an attribute-aware in-order scheduler. When a data block is processed for output, an *attribute* is added to the block defining its order in the application-defined attribute space. When the DataStager services its request it guarantees that request i will not be processed before request j if $i > j$. Thus, when a request is processed, the DataStager already knows the sizes of all previous requests and can write a shared file without any additional synchronization. In the case of multiple DataStagers addressing a group of requests, the sizes can be exchanged within this small group of nodes, or multiple shared files can be created and merged in an additional processing step.



(a) Using 4 staging nodes with GTC (b) Using 16 staging nodes with GTC

Figure 5: GTC run time with multiple data extraction strategies.



(a) Using 4 staging nodes with GTC (b) Using 16 staging nodes with GTC

Figure 6: Percentage overhead of data extraction on application runtime.

4.3.4 Rate limiting scheduler

Phase aware scheduling provides solutions for applications that follow regular predictable patterns for data output. Using these predictable patterns we can, with a degree of confidence, avoid the resource usage conflict between DataStager and intra-application communication. In the case of applications with irregular patterns, however, such as AMR applications [67], the state-aware scheduler cannot predict the phases of the application with any reasonable degree of accuracy. In such cases, a different strategy must be employed.

A rate limiting strategy for extracting data from a large cohort of application nodes can be considered if the periodicity of data output is sufficiently large and if the data is

not required to be processed immediately. By managing the number of concurrent requests made to the application nodes, the DataStager can greatly reduce the impact of perturbation on intra-application communication. Limiting the rate can have performance implications in terms of reduced ingress throughput and slower time to completion for the requests. This impact can be avoided by appropriately varying the level of concurrency to achieve a proper balance of throughput and perturbation.

Consider an application that writes out data of size 200 GB every 5 minutes from 1024 cores. In order to reduce perturbation and maintain a consistent drain rate from the application, we need to manage the level of concurrency of requests. As seen in Section 4.4.1, by varying the number of staging nodes, we can control the ingress throughput from the application. Using 128 compute processes per DataStager nodes, we see ingress throughput of approximately 8GB/s. Thus, draining at the best possible speed we can complete the data transfer in 25 seconds. However, this may result in an unacceptable level of perturbation on the source application. By reducing the number of concurrent data transfer requests being serviced to 1 per DataStager, we would increase the time to completely move the data from the compute nodes to the DataStager, but we also could reduce the impact on performance caused by background I/O.

One aspect of the rate limiting scheduler is the determination of an appropriate concurrency rate for each type of data output by an application. Currently, we do not modify the rate autonomically, but we are investigating policies that will enable the DataStager to determine the optimal rate at which data is extracted.

4.4 Evaluation

We developed and evaluated the DataStager on National Center for Computational Sciences (NCCS) Jaguar Cray XT at ORNL. At the time of this experiment a single Jaguar node was a 2.1 GHz quad-core AMD Opteron processor and 8 GB of memory, connected to the Cray SeaStar2 interconnect. The interconnection topology is a 3-D torus with each

SeaStar2 link enabling a maximum sustained throughput of $6.5GB/s$. The compute node operating system is Compute Node Linux (CNL), and all applications were compiled with the pre-installed PGI compilers. Cray uses the low level Portals API for network programming and provides high level interfaces for MPI and Shmem.

As mentioned before, we used the Gyrokinteic Turbulence Code, GTC [60] as an experimental testbed for DataStager. GTC is a particle-in-cell code for simulating fusion within tokamaks, and it is able to scale to multiple thousands of processors. In its default I/O pattern, the dominant I/O cost is each processor's output of the local particle array into a file. Asynchronous I/O potentially reduces this cost to just a local memory copy, thereby reducing the overhead of I/O in the application. No effort was made to optimize the location in the interconnect mesh of the compute processes with regards to the DataStagers.

We also performed micro-benchmarks to evaluate the maximum throughput for data extraction. For all tests we used the NCCS Jaguar platform with the number of client nodes varying from 64 to 2048. The DataStager nodes used the entire physical node - 4 cores and 8 GB of memory.

We evaluated 6 different data extraction scenarios:

- **CD** is the *continuous drain* method for data extraction.
- **PA** is the *phase aware* method to manage the timing of data extraction.
- **Con_X.** is the *rate limiting* scheduler which limits the number of outstanding concurrent requests to X. We explored two values for X, 1 and 4.
- **PA_Con_X.** is a stacked combination of the *rate limiting* scheduler and the *phase aware* scheduler. In this scenario the number of concurrent requests are limited to X and a new request is only issued if the application is the compute phase. As above we explored two values for X, 1 and 4.

For the remainder of this chapter, we use the above notation to reference the data extraction strategies.

4.4.1 Ingress throughput evaluation

To measure the ingress throughput we use a parallel test application writing out 128 MB per node per output. Each client process issues an output request and waits for completion immediately. The time taken to complete the data transfer for all client processes is used as the measure for maximum ingress throughput to the DataStagers. In order to maximize the ingress throughput, we only utilize the continuous drain scheduler and retire the data buffers from the DataStager staging area immediately.

As can be seen Figure 4, the ingress throughput increases as we increase the number of staging nodes. For a single staging node, we see an ingress throughput of 1.2GB/s. As we increase the number of staging nodes the available data extraction throughput increases to more than 55GB/s for 128 DataStager nodes with 2048 client processes. Note that the ingress throughput does not scale arbitrarily with addition of more DataStagers for a small compute partition size. The experiments here show that as we increase the total count of the clients moving data to the DataStager, the aggregate ingress throughput increases until limited by the interconnect capacity.

4.4.2 GTC benchmarks

We have extended I/O in the Gyrokinetic Turbulence Code GTC [60] using the ADIOS application interface. The flexibility of the ADIOS interface allows us to run experiments using blocking binary I/O, DataTap with multiple scheduling strategies, and even no I/O without modifying the application binary. For all of the runs, the total configuration size was adjusted so that the amount of data per compute node was a consistent 6,471,800 ions/core. GTC is used for evaluation due to the size of the data output as well as the ability for the code to scale to more than 30,000 cores. We have used a version of the GTC source tree with support added for ADIOS as its I/O library. Using ADIOS has provided us with the opportunity to perform exact comparison tests with the application by simply switching a parameter in the *config.xml* file. To allow better understanding of

the performance of different scheduling parameters, we disabled all outputs from GTC except the particle output. Also, to keep the comparison to multiple run sizes as close as possible, we used *weak scaling* (i.e., maintain a fixed per process problem size) instead of *strong scaling* (i.e., maintain a fixed global problem size and only change the number of processes) to maintain a consistent size for the output data per core. Thus, the total size of the problem increases but the size per core remains constant. The data size from each output is 188MB/core. The total data volume varies with the number of parallel cores from 12 GB/output to 3.8 TB/output. The output had a periodicity of 10 iterations (approx. 3 minutes wall clock time) and the application ran for 100 iterations. In order to avoid the variations at startup we only measure the time from the 20th iteration onwards.

4.4.2.1 Runtime impact from background staged I/O

We compare the GTC run time for each of the different data extraction methods described earlier. In order to get an accurate understanding of the cost associated with I/O, we also evaluate the default POSIX data output (through the POSIX transport method in ADIOS) as well as a no-op transport method, NO-IO. When required, we also use the default implementation of the MPI-IO transport method as a second example of synchronous I/O.

Consider Figure 5(a), which shows how application run time is impacted by different DataStaging schedulers. For a small number of compute cores (e.g. 64), there is very little impact on the overall performance from I/O. Con_1 is the only strategy that shows appreciable overhead and even then it is less than 5%. At 512 compute cores, the different schedulers start to differentiate. 512 is the minimum size at which we see significant impact from perturbation. Below 512 compute cores, the runtime difference between synchronous and asynchronous I/O is minor. As we move to 512 and 1024 cores, we see statistically significant differences in run times. PA, PA_Con_1 and PA_Con_4 continue to show very little overhead from I/O extraction. Con_1 and Con_4 perform at the same level as synchronous output with POSIX. The low impact of all asynchronous strategies is also evident with 2048

application cores. The performance impact of POSIX increases to over 20%. Con_4 and PA_Con_4 maintain an acceptable level of performance impact even at this scale. The same pattern is observed with 16 staging nodes, but the performance impact of background I/O using continuous drain increase greatly.

In Figure 6, we quantify the percentage cost of using the DataStager for performing non-blocking I/O for 4 and 16 staging nodes. For a small number of client cores (64), the synchronous POSIX method offers superior performance. However as we scale and the total size of the data increases, the time spent in synchronous I/O increases more than the overhead of the DataStager method. The percent impact of the POSIX method increases from less than 2% with 64 cores to 10% with 1024 cores, increasing to over 25% with 2048 cores. In contrast, the impact of the DataStager depends greatly on the type of scheduling mechanism used, as well the number of staging nodes used stacking the rate limiting scheduler with the Phase aware scheduler PA_Con_4 and PA_Con_1 provides the best performance as we scale the number of compute cores. The number of staging nodes also has an impact on the perturbation of the compute application, with 4 staging nodes showing lower perturbation in general than 16 staging nodes.

4.4.2.2 Breakdown of impacted subroutines in GTC

To further our understanding of the performance characteristics for the DataStager, we analyzed the runtime for the *smooth* subroutine in the GTC code path. For clarity, we are not displaying the impact on the rest of the subroutines. The function *smooth* immediately follows the data output and hence, in cases of improperly managed data transfers, shows the greatest level of perturbation. Consider Figure 7, which shows the runtime for the smooth and restart function with 1024 cores for all schedulers as well the synchronous methods. For both POSIX and MPI-IO, we see a large increase in the time for restart, signifying the I/O blocking time. We also see a significant increase in smooth due to the partial buffering of output data by the Lustre client and subsequent background evacuation

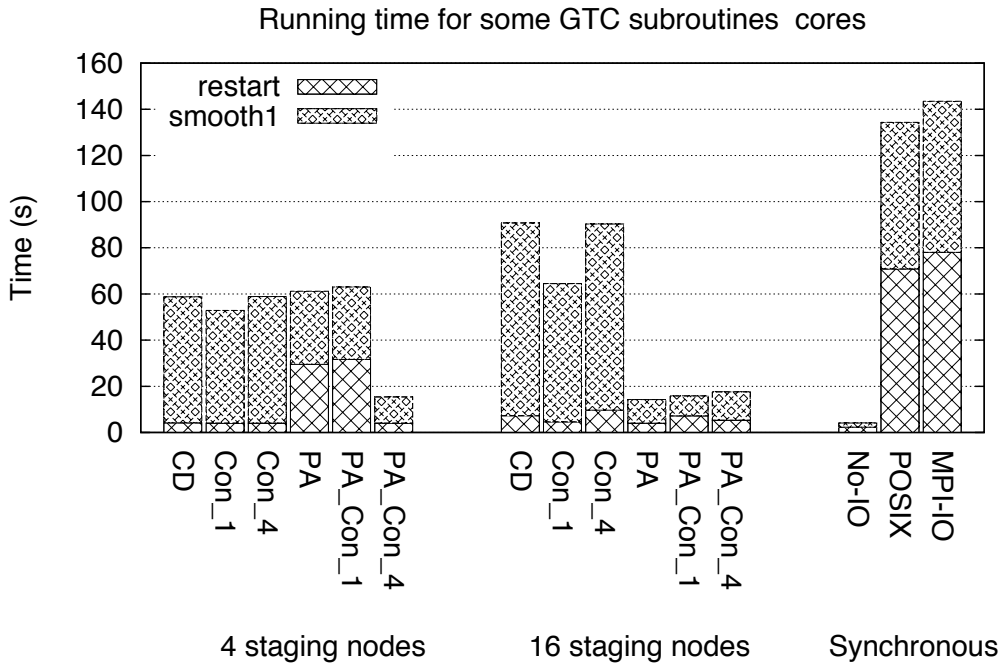


Figure 7: Total Runtime (8 iteration) for functions impacted by background I/O with 1024 cores.

of the buffer to OSTs. In contrast, the DataStager shows very low runtime overhead for the restart subroutine. However, the performance of *smooth* is negatively impacted by non phase aware data extraction strategies. such as CD, Con_1 and Con_4.

4.4.2.3 Time to complete data extraction to DataStager

One important factor to consider for asynchronous I/O is the total time taken to service all of the application's I/O requests. This time determines how often an application can issue an asynchronous I/O request without requiring additional buffer space. We compared the completion time for the I/O phase at 1024 application cores (total data size of 180 GB) and the results are shown in Figure 8. As CD tries to extract the data as fast as possible, it is not surprising that the time taken by CD is the lowest for 4 staging nodes, and next to lowest for 16 staging nodes. However, both Con_1 and Con_4 also show very

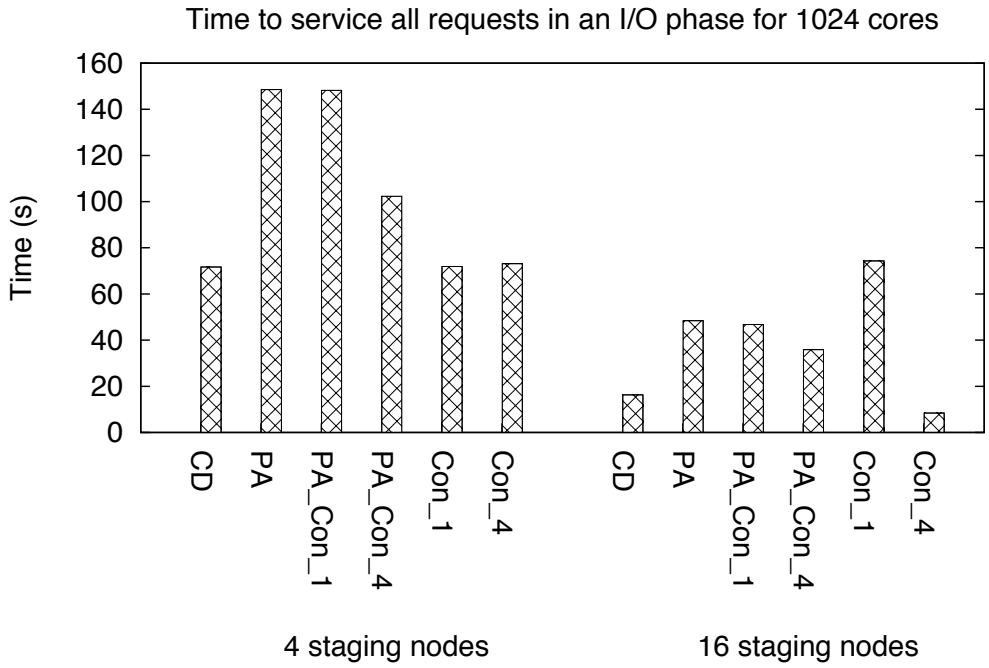


Figure 8: The time taken to complete all pending I/O requests from all processors.

low completion time. This is because by limiting the concurrency of the inflight requests the ingress throughput for a single request is maximized. The phase aware strategies show much higher completion time with PA and PA_Con_1 performing almost the same. This is because phase aware strategies can only initiate a transfer during a small window in order to avoid interfering with the application.

4.5 Discussion

The performance characteristics of the DataStager-DataTap transport serves as the basis of our implementation of data services. The ability to utilize the scheduling framework, which not only reduces the impact of asynchronous I/O in large scale environment, but also provides a framework for performing data attribute based scheduling.

However, data services also require the execution of application and user specific functions. As mentioned previously, the set of functions supported by data services can reap significant benefits from optimizing the location at which these functions are executed. In Figure 2 we provide the heuristics which can be used to select the *right* placement of these functions. The dynamic movement of these functions and the flexibility in selecting their placement, has been enabled by the mechanism of the JITStager, described in the next chapter.

CHAPTER V

JUST IN TIME STAGING

We have previously addressed the I/O bottleneck in high end machines in multiple ways. We have utilized ADIOS, a componentized I/O framework, which can be used to optimize I/O on a per-platform and per-application basis, including for shared storage targets[49]. A complementary solution integrated with ADIOS is data staging [3], where data is moved to storage in three distinct steps, Figure 9. The first step buffers data in the application's address space. This requires a memory copy for data, but its ability to transfer larger data blocks outweighs these copy costs. The second step sends a request to the staging area, informing it of the availability of filled output buffers. The staging area itself is comprised of a set of additional resources designated for data management tasks. It can be seen as a large, transient memory buffer or cache that provides performance insulation from storage system bottlenecks. In the third step, the staging node uses a remote read operation to transfer the buffer into staging memory. By switching the control of the transfer from the compute application, which is only responsible for issuing the availability notification, to the staging node, asynchronous data movement is achieved. The staging area can independently carry out its own data scheduling and buffer management actions. The final step, then, moves data from the staging area to disk and/or to remote machines or storage facilities.

While data staging has been found to ameliorate the I/O problems of high performance machines, a further shift is necessary in how I/O is performed. To address the I/O needs of future high end applications, we have moved from synchronous I/O on compute nodes, to asynchronous I/O via data staging areas. Next, we must move from the 'cache' model

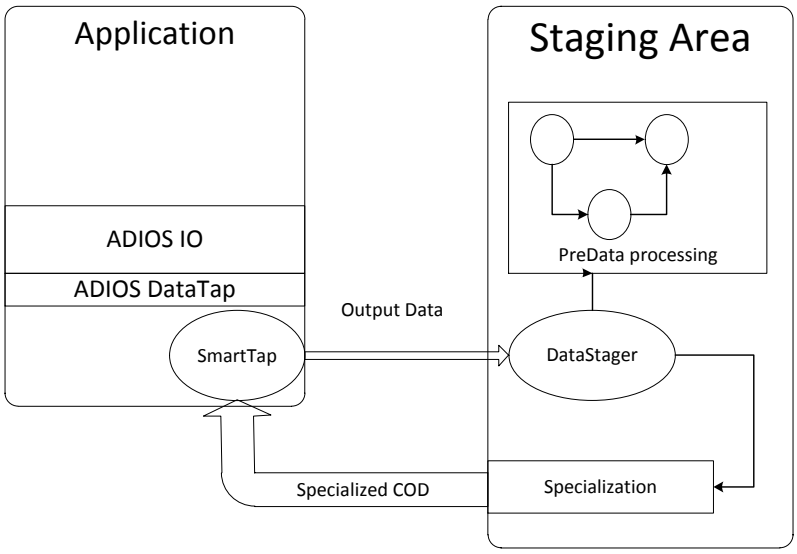


Figure 9: The JITStager architecture.

of data staging to a generalized 'computational I/O' model that also leverages the substantial compute capacities of staging nodes. The slack computational capabilities of the staging area can be used for data indexing, data sorting, and data reorganization. Given the data deluge in the current and next-generation hardware environment, this serves to provide users with quicker insights into the data produced by their applications and/or to better prepare such data for subsequent deeper analysis or visualization. This can provide a significant performance boost for analytical workflows operating on stored data, and, moreover, portions of these workflows can be moved into the staging area for near-time analysis and monitoring. In fact, with the LIVE system [1], we have already shown how near-time data processing can be used for application output validation and visualization, and in [75], we have introduced the notion of PreDatA analytics to characterize and demonstrate useful near-time data processing actions.

The JITStager software system presented and evaluated in this chapter extends our prior work *by applying computations to I/O actions along the entire I/O pipeline*, starting with

the output actions performed by the application, to staging areas, and beyond. Namely, JITStager permits the user to execute operations on generated data not just in the staging area, but also at the points of data generation (i.e., in the compute application). This leverages the aggregate computational resources of compute nodes to accelerate important I/O processes and, more generally, provides increased flexibility (and in some cases performance) in how data movements and manipulations are carried out. Using dynamically generated binary code, we avoid the performance pitfalls and potential OS-dependencies normally associated with moveable code fragments while maintaining the flexibility advantage. This capability is further extended to include functional specialization using information gathered within the staging area, allowing for an even greater opportunity for performance optimization. A typical use case is one in which features of interest in data are discovered while the application is running. These then cause output specializations that make it easier to capture and understand these features at scale and for any output steps where they occur.

The JITStager system has three distinct parts. The **SmartTap** is an asynchronous buffered data transport module that replaces the traditional file output layer. In order to achieve deeper penetration, we have developed SmartTap as a transport method within ADIOS [49], thereby simplifying the task of switching to SmartTap as an output method. SmartTap also serves as the execution engine for data customization operators. A detailed description of SmartTap can be found in Section 5.4.

The second part of the JITStager system is the **DataStager**. The DataStager manages the transfer of data from SmartTap, utilizing both in-built and user specified schedulers to minimize the interference from background I/O operation. Similar to SmartTap, DataStager also serves as an execution engine for data customization operations. Once the DataStager has completed the transfer and customization, it forwards the data to data processing pipeline within the staging area. We discuss the DataStager in Section 5.4.

Finally, JITStager includes the PreData functionality for data processing and output that completes the data processing steps required and produces output in the required data

format.

JITStager's technological contribution is a set of software with which end users can, at runtime and wherever needed, place select computations directly into the output actions taken by their high performance codes. Stated in terms of the data staging system we have presented earlier [3], I/O actions can be changed at any time (including in specific or select compute nodes) using SmartTap computations to apply lightweight and efficient manipulations to the data being generated. Extending the computations and data acquisition actions performed by the extremely scalable deployment of SmartTaps within compute nodes, JITStager then applies subsequent data manipulations in the staging area through PreData.

JITStager's novelty lies in the way in which data manipulations are efficiently and dynamically associated with I/O actions. Most existing output systems either require end users to change their applications to output exactly the data they need [19], or they rely on applications themselves to implement rich output strategies [62]. Also available to end users are service-oriented approaches to data output in which specifications and software separate from applications select the actual data to be output by applications [49]. The former may require end users to re-compile and re-validate their codes when output actions are changed. The latter assemble data in raw form in some output buffer, then move it to where additional manipulations can be performed.

JITStager improves on such work by providing I/O methods that (1) preserve the independence property of service-based data output, (2) can extend I/O actions at any time and in any place needed, and (3) are sufficiently high performance to go beyond providing useful new functionality to also potentially substantially improving output performance. In particular:

1. SmartTap can use CPU cycles on the node performing output to 'make data right' immediately, thereby avoiding subsequent and potentially memory- or communication-intensive steps that re-format or re-organize data. In this fashion, we leverage the

widening gap in processor vs. memory performance of modern computer architectures. SmartTap can also be used to better annotate and label data, by adding attributes to it, thereby facilitating later steps in the output pipeline. SmartTap actions can also reorganize data, again to facilitate later data manipulation or storage actions, including those required for data shared across multiple interacting computational models[24, 2].

2. SmartTap code is safe, efficient, and easily changed. It is efficient because it is generated directly into memory as a native binary instruction stream. It is safe because it is generated in ways that guarantee that SmartTap codes cannot affect applications, thereby eliminating the need for application re-validation. Such isolation is attained without the need for hardware support [35], using techniques like those described in [33, 70]. It is easily changed, because SmartTap actions can be deployed whenever or wherever users desire them to be used, thus making it easy to change what data is output and how it is output. These changes can be made whenever needed by later analysis steps, and/or by end users.

JITStager functionality can be used in many ways. First, because the decisions concerning I/O pipeline customization are made in the staging area, the global information about output data can be used to dynamically explore or focus on certain data features of interest to end users. Second, data can be better partitioned to allow for load balancing (in terms of space and/or computational load) across staging nodes. One example might be a customized pipeline that breaks up a large, distributed array of atomic locations and redistributes it to a collection of staging nodes for a multi-viewport visualization (i.e. top, bottom, left, right) so that each node only has to render the atoms that one could "see" from that vantage point. Third, data can be specialized not only in terms of structure or organization, but in addition, it is possible to dynamically change select content of the data being

output. Examples include data selection, filtering, and transformation. A typical usage example is one where a coupled model not only requires data to be organized differently, but requires data to be summarized or interpolated [38].

5.1 Related work

Scalable I/O performance for HEC platforms has been studied extensively, resulting in multiple approaches to solving the performance bottlenecks explored. Scalable file systems [18, 66, 46], I/O middleware [74, 40, 63], and I/O component libraries [49, 47] address the needs of the scientific application by optimizing specific scenarios or providing optimized I/O methods. Compared to these methods, JITStager proposes a significantly different, 'computational' model for dynamically user-customized I/O.

Staging infrastructures such as I/O delegates [57], I/O forwarding [6, 36], and DataSpaces [25] are the closest analogues to JITStager. However, JITStager provides additional resources for computation within SmartTap, and it allows for computation within the staging area.

MRnet [65], part of the Paradyn[53] project, is a scalable mechanism for performing aggregation within the compute partition. MRnet provides functionality similar to JITStager by allowing additional computation within the compute partition, but it does not address the complex computational customizations that JITStager offers.

Map-Reduce [30, 21] posits a model similar to PreData and to some extent, JITStager, for analytical computations utilizing *map* and *reduce* operators, but PreData and JITStager support both more general models of parallel computation and allows for a more complex interactions between the reduce operation through function specialization.

5.2 Customizing the data pipeline

We demonstrate the performance and flexibility of JITStager on two commonly used types of applications. Warp is a molecular dynamics application, a predecessor to LAMMPS [62]

that our local collaborators have extended for particular force field calculations. It is configured to use a spatial decomposition of particles and produces a number of outputs for analysis and snapshots. GTC [41] is a particle-in-cell (pic) code simulating the plasma in a Tokomak reactor. While both applications are scalable we use the Warp application as a testbed on our linux cluster, upto 128 cores, while we utilize GTC on ORNL's Jaguar XT5 for test sizes greater than 8K cores.

To demonstrate the functionality and evaluate the performance of the JITStager customization pipeline, we have developed prototype I/O customization operations based on feedback from the users of these applications. Although these customizations are specific to the experimental requirements of the user, we find that they show sufficient generalization to demonstrate the advantages of a dynamic customizable approach to I/O pipelines.

1. **Data Filtering**. Both GTC and Warp process particle positions and velocities and likewise the large portion of their output data sets is particle data. For very large output data sets, scientists will often seek to reduce the raw data to regions of interest. However, due to the evolution of the simulation, the regions of interests vary throughout the lifetime of the application and furthermore are highly dependent on the experiment. A simple example of data filtering is a bounding box filter, the constricts the output data to a specific 3d region. Such a filter is common for visualization of molecular data and we evaluate the bounding box filter with Warp. GTC, a pic (particle-in-cell) code, requires the data filtering operation to restrict the output to a specific 2d plane

2. **Statistical Tagging**. Data output from the application is a collection of variables pre-determined by the application developer. While the user can discard specific variables in the final output, or turn off the data output altogether for more applications, it is usually not possible for the user to enhance the output with new information such as statistical characterstics. JITStager seeks to alleviate this issue by allowing

```
filter:x:{
  for(i= 0; i < element\_count; i=i+3) {
  if(input.x[i] > bbx2 && input.x[i] < bbx) {
  if(input.x[i+1] > bby2 && input.x[i+1] < bby) {
  if(input.x[i+2] > bbz2 && input.x[i+2] < bbz) {
    output.x[j] = input.x[i];
    output.x[j+1] = input.x[i+1];
    output.x[j+2] = input.x[i+2];
  }}}}
}
```

Figure 10: Example data filtering operator using the buffered sampling method.

the output data to be preprocessed and annotated by relevant statistical information. These annotations can be added within the compute node utilizing the SmartTap, or within the staging area, or added to the output in a post-processing step after the data has been written to storage. There are distinct advantages and disadvantages to annotating the data at each of the three stages and we evaluate these tradeoffs in Section 5.5.2.

3. **Statistically Relevant Sampling**. In addition to data tagging, data pipeline customization enables the use of statistically-relevant sampling for data reduction. For example, consider a study of the behavior of a fluid near the critical point of freezing. A statistical analysis of the particle velocities is relevant, but behavior is dominated by the fluctuations out in the "long tail" of the distribution. Generating output with a histogram representing the probability distribution in the core, but having the tails represented by exact counts would be more accurate for most such studies. Not only is the data volume reduction for this operation significant, thereby providing a strong use case for the development of intelligent filtering operations, but it also improves the immediate utility of the data. This example also demonstrates the dynamic specialization requirement in concert with global feature extraction on the staging area. Because the customization is performed on global data features, the global features must be reduced from SmartTap annotations within DataStager and the specialized C-o-D function pushed back to SmartTap, as shown in Figure 9.

```
sample:x:{
  for(i= 0; i < element\_count; i=i+3){
  if(input.x[i] > bbx2 && input.x[i] < bbx) {
  if(input.x[i+1] > bby2 && input.x[i+1] < bby) {
  if(input.x[i+2] > bbz2 && input.x[i+2] < bbz) {
    FFSMarshalArrayElement(i);
    FFSMarshalArrayElement(i+1);
    FFSMarshalArrayElement(i+2);
  }}}}
}
```

Figure 11: Example data filtering operator using the inline sampling method.

5.3 Time to data - TTD

In our previous work with PreData for preparatory data analytics, a specific metric that appeared is "Time to Data" or TTD. We define "Time to Data" as the time required for the generation of data by a simulation, the time taken to output data from the simulation, the time required for the preparation of data for analysis and the final output to storage before the analysis workflow takes over. Thus, TTD defines the latency from the application to the point where data is ready for the analysis pipeline.

TTD is an important metric for evaluating the availability of data for users of high performance applications. Therefore, it is also an important metric in evaluating any I/O customization pipeline and is highly dependent on the structure of the PreData pipeline. Alongside TTD we must also consider the consumption of additional resources utilized by the I/O pipeline. In this chapter, we only consider the number of CPUs and the time they were occupied as the resource. This formulation satisfies the needs of JITStager, but resources can also be measured in terms of other factors such as total memory occupied, power consumed, I/O utilization or a combination of these metrics. The optimization process is determined both by the underlying architecture of the client system as well as specific user requirements. To illustrate, consider as one extreme example an I/O pipeline that only generates and outputs data from the application to disk and uses a single process to read the data from file and prepare it for the analysis workflow. Although the TTD for

this structure is extremely high, the resource consumption is minimized to a single additional process. The lesson from this simple example is that in general, there is a balance to be reached between TTD and resource consumption.

5.3.1 Placing JIT data specializations

An important property of JIT data output specializations is, the importance of performing these operation at the *right* location within the output pipeline. An operation such as a filter based on a bounding box will have greater performance impact when it is performed 'closer' to the source of the data, for instance, by reducing data volume at the compute node and before the data is moved to the staging area. Such filters help obtain several beneficial performance properties: (1) the network throughput requirements for the I/O operation are reduced and hence the performance impact on the application can be minimized; (2) the buffering requirements on the staging area are also reduced, providing better performance insulation for I/O interference; and (3) since the data is already filtered, the analytical operator does not need to process irrelevant data blocks.

Performing specialization at the source of the data is not always appropriate, however. Specifically, while operations that filter the data based on user parameters work well within the compute area, scalability is questionable for operations that require collective operations. For example, the statistical sampling operation requires knowledge of the global properties of the data. In order to optimize this operation, there has to be an intermediate step for creating the specialization operator that can extract global parameters. The staging area is ideal for performing this task, since all data is directed through the staging nodes. Computing the global bounds on the data and then pushing the specialization back to the compute nodes is the more efficient way of handling this task.

Finally, combining source- and staging area-level operations, an ordered stream can be obtained by first tagging the stream with attributes, at the sources, and then using a user-specified attribute scheduler that operates entirely within the staging area. Utilizing

such server-side scheduling of the data reduces both communication overheads within the staging area as well as the latency to disk for data blocks.

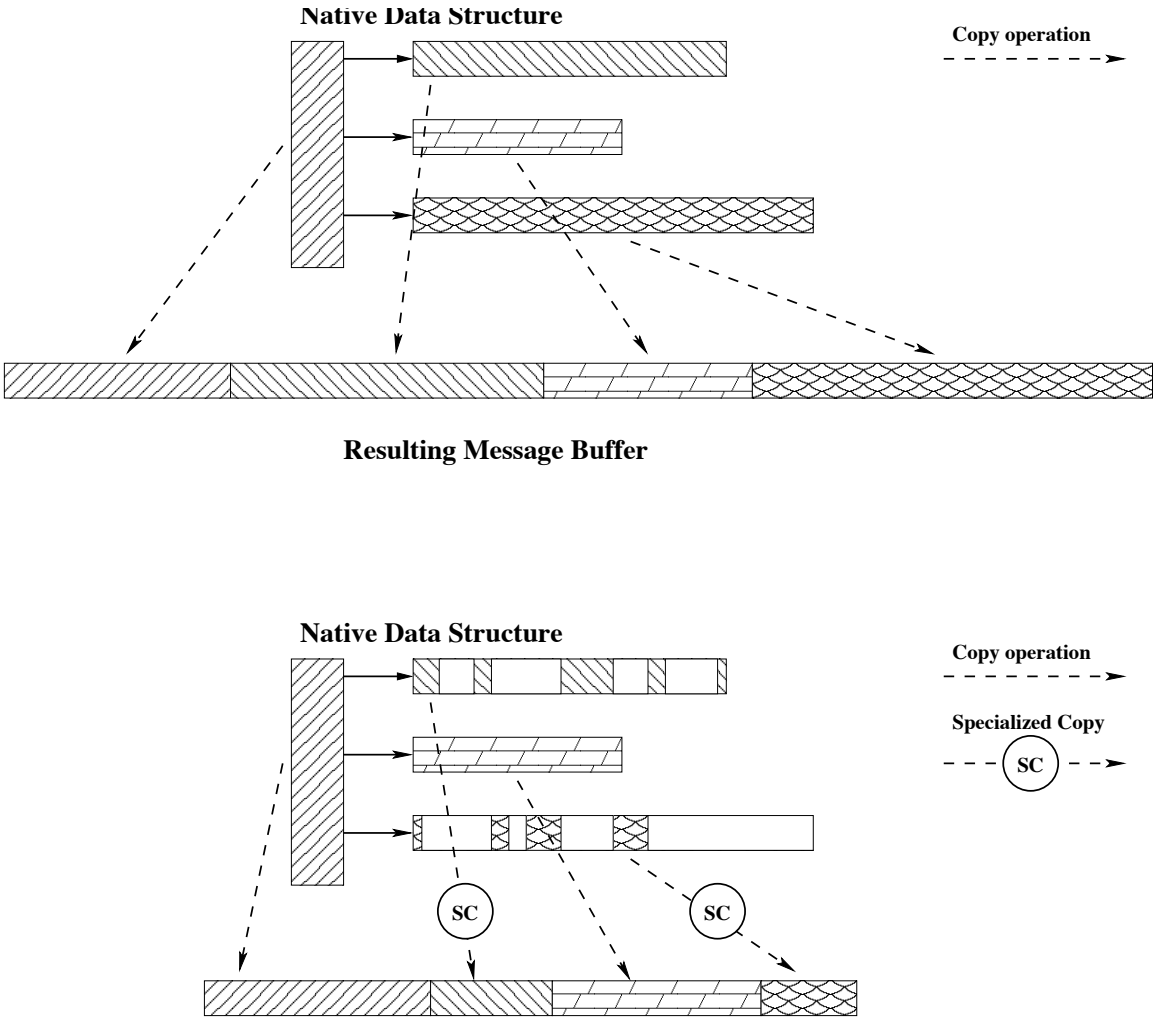


Figure 12: DataTap Marshalling.

1. *The global feature extraction module.* In order to efficiently and intelligently introduce specialization functions into the data pipeline, we have to recognize the properties of the extracted data, often within a global scope. For example, consider the statistically relevant sampling specialization mentioned previously. To successfully construct a probability density function for the data, we need not only the boundaries within which to utilize sampling, but also the extent of the size of the bins used for

the sampling function. Although data is output only from the compute application, the computation of these global features requires either additional global computation within the application, or it needs the software offered by JITStager to efficiently extract global features from the output data. Once these features have been extracted, we use the specialization management module to create and push back to the compute application the specialized sampling function.

2. *The specialization manager.* Along with global feature extraction, the JITStager architecture requires an external control module that can based on either extracted features or on user-specified parameters, creates the specialized functions used for data processing within the pipeline.

3. *SmartTap and DataStager, the specialization processing components.* Once the specialized function code has been generated, JITStager allows the output pipeline to use the generated functions for in-transit data processing. Data processing in this case can be performed either within the compute application, using the SmartTap, or within the staging area, using DataStager. The placement of these functions is a matter of policy that is handled by the management module.

5.4 Implementing an efficient I/O pipeline

The details of the data processing components, SmartTap and DataStager, have a strong influence on the capabilities of the JITStager system.

5.4.1 FFS and C-on-Demand

Both SmartTap and DataStager use well-defined binary data formats to represent and organize the data output by applications, employing an internal file/data format termed FFS, an extension of our prior PBIO work[15]. During normal operation, FFS uses dynamic code generation to improve message decode/unmarshal performance. In particular, because the precise layout of incoming FFS-encoded data depends upon the architecture details of the

sender, FFS does not assume *a priori* knowledge of the layout and instead generates a customized unmarshalling subroutine for each incoming message layout. These subroutines are reused for each subsequent message that shares the same layout and are generated using the Georgia Tech-developed DILL package that provides dynamic code generation via a virtual RISC instruction set. DILL currently can generate code for x86, x86-64, ia64, sparc, arm5 and mips architectures. Another part of FFS is C-o-D (C-on-Demand), which implements a subset of C, is a relatively thin compiler layer built on top of DILL, consisting of a lexer, parser, semantics, and code generation. C-o-D's design supports extensibility in that types, structures, external variables, and subroutines can be made available to the generated code, without C-o-D being aware of those items at compile time.

C-o-D is used in both SmartTap and in DataStager as an efficient mechanism for customizing data handling. As described below, C-o-D is used to directly manipulate data in both SmartTap and DataStager. It is also employed in SmartTap to affect data marshalling in a novel way.

5.4.2 SmartTap

SmartTap is an extension to the DataTap asynchronous data movement library designed for low overhead data extraction. Unlike DataTap, SmartTap's current implementation is not completely asynchronous. Rather, the C-o-D-based functions associated with SmartTaps are executed inline with data output paths. This allows the staging area to push back C-o-D code fragments that can be used for further specialization of the output data, producing outputs that better meets the needs of subsequent processing steps, including those of consumer workflows written with, for example, Kepler.

SmartTap provides two major technical innovations. First, because SmartTap is built on C-o-D it allows an external manager/controller to determine – through iterative methods – the best locations for introducing in-transit data processing. Second, SmartTap extends the C-o-D paradigm to merge select operations with the action of marshalling into a buffer.

This method of reducing data can have a significant impact on the performance of large data output sets and produces targeted data output that can be easily used within further stages in the data operational pipeline. However, since SmartTap executes within the compute nodes itself, there are some limitations to the functions that can be executed within the SmartTap. We discuss those limitations in more detail in our experimental evaluation, but nonetheless, the specialization of C-o-D functions allows some complex functions to be simplified and optimized for execution within the SmartTap. For functions that do not translate to SmartTap characteristics, the specialization manager is responsible for selecting the specialization target.

As mentioned above, SmartTap uses C-o-D in two specific ways to customize data staging, each with specific advantages which are evaluated in Section 5.5. First, SmartTap uses C-o-D to filter/transform/subsample the extracted application data before that data is passed to FFS for marshalling. This relatively straightforward approach, called *filter-then-marshall*, allows C-o-D to operate on the entire extracted application data structure at once. However, in order to keep memory management complications to a minimum, applications of *filter-then-marshall* that involve changing data require a copy to be made of that data.

SmartTap uses FFS to format all data placed into its output buffers, but an additional novel contribution is its ability to customize such marshalling behaviour using subroutines generated with C-o-D (C-On-Demand). This SmartTap technique is called *customized-marshalling*.

To better explain how customized marshalling can be used to efficiently implement SmartTap customization functionality, we first considered basic marshalling support supported by FFS. FFS marshals complex data structures, including pointer-based structures. A common way to gather data items for transport is to create a base structure which includes pointers to other data items. During marshalling, FFS packages the entire structure, including the base structure for transport to the destination host. This marshalling involves copying the base data structure into the message buffer, calculating the sizes and message

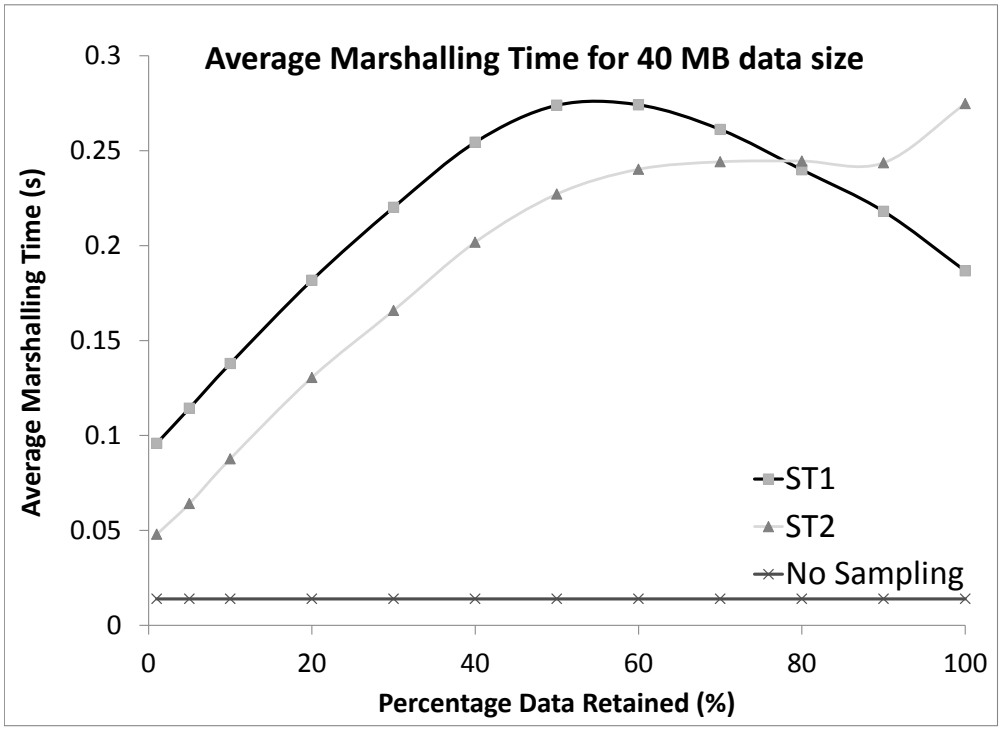


Figure 13: Cost of marshalling data for transfer for different sampling sizes, data size = 40 MB.

buffer locations of each sub-element, overwriting the pointer values in the copied structure with the message offset of each subelement, and then recursively applying this marshalling procedure in order to move the transitive closure of all subelements into the message buffer. This operation and its result are shown in the upper half of Figure 12.

In order to accomplish the customized marshalling desired by SmartTap, we modified the procedure described above so that a C-o-D-generated subroutine controls the copying of particular subelements. In particular, we have modified the FFS marshalling procedure in order to support the use of a specialized copy routine (show in the lower half of Figure 12. This copy routine, supplied in textual source form by the specialization policy manager, and subject to dynamic code generation before use in SmartTap, can examine the data being marshalled in order to make individual decisions about marshalling particular elements.

5.4.3 DataStager

The DataStager previously described in [3] and Chapter 4 is the staging mechanism used in concert with SmartTap. DataStager provides users with server-side scheduling for data movement, a structured data format which supports reflection, and the infrastructure for deploying online PreDatA analytical operation.

In JITStager, the DataStager is further extended to include functionality for the deployment of code from external sources. As described previously, the use of C-o-D allows the deployment of fast dynamically generated binary code, without suffering from the performance bottlenecks of interpreted code or the lack of flexibility of dynamically loaded libraries. Furthermore, these functions can be further specialized to the experimental scenario and user's requirements.

DataStager services the application requests for data extraction in the order determined by the scheduling mechanism. Once an extraction request has been completed, the resulting data is queued up within the JITStager server for user-specified processing. In addition to performing the PreDatA analytical processing, DataStager also passes the data to the global feature extraction module. This module employs user-specified feature extraction functions across the entire JITStager cohort. One advantage of this approach is the increased scaling offered to the application by off-loading these collective operations to the smaller set of staging area nodes (typically operating at a ratio of 1 staging node per 256 compute nodes). A functionality argument is the flexibility offered by allowing the specification of these functions at run-time, instead of tightly coupling them with the application.

Once features have been extracted successfully, they are handed off to the specialization policy manager. This module generates customized C-o-D functions that are then used to customize the data pipeline. Specialized code is pushed up the data pipeline to either the SmartTap or to operate in the staging area.

In addition to computing on the data, DataStager offers the functionality of an attribute scheduler. The attribute scheduler was originally defined in [3] to be a client-controlled

mechanism for performing scheduling decisions that conform to the user requirements. A simple example of the attribute scheduler is extracting a stream of data where all blocks are in order based on node id. Extending this scheduler is trivial due to JITStager's capability of runtime insertion of dynamic C-on-Demand code into the scheduler stack. The use of the attribute scheduler can bring down average storage latency when writing out contiguous files as well as reducing the memory required for buffering within the staging area.

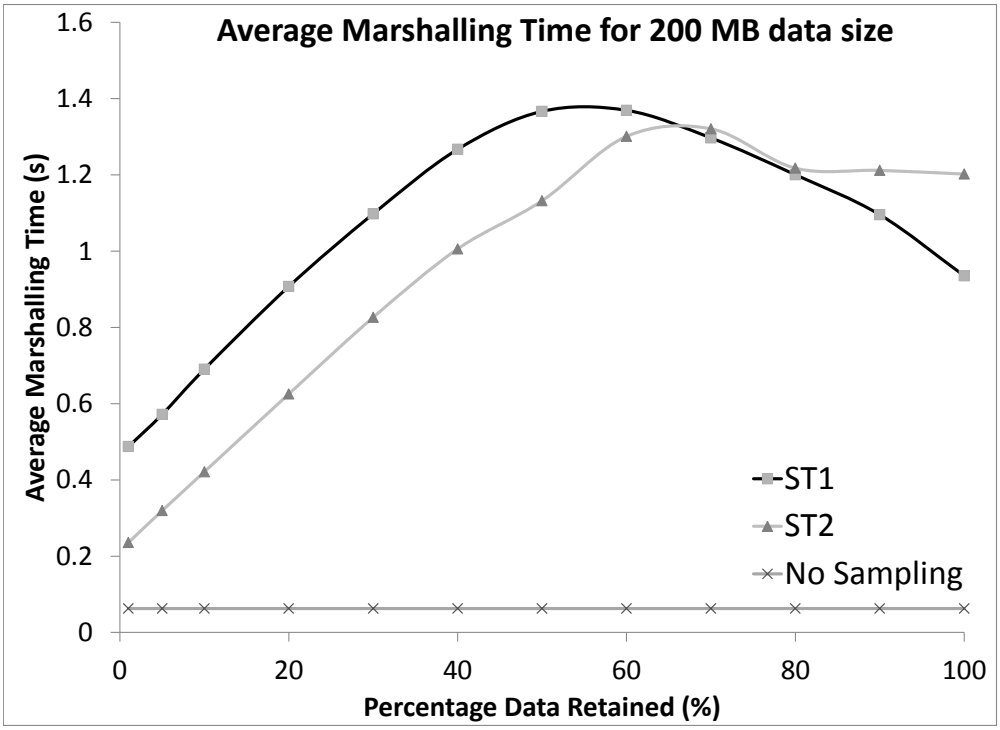


Figure 14: Cost of marshalling data for transfer for different sampling sizes, data size = 200 MB.

5.5 Experimental evaluation

The JITStager system is designed to address both the functionality requirements of modern science as well as its performance needs. We evaluate JITStager in two parts. In the first part, we break down the different performance considerations of customizing the data

pipeline by evaluating the impact on marshalling time for different data retention percentages. We also compare the data customization cost when customization is addressed by the SmartTap to the cost when it is handled in the DataStager.

Next, we utilize the Warp molecular dynamics application described in Section 5.2 to evaluate the aforementioned customization scenarios. For the data reduction scenarios, limiting the output to user specified bounding boxes and for utilizing statistical sampling, we look at the overall impact on the application running time, the latency to storage, and the reduction in data size.

Throughout the evaluation for Warp, we refer to the *filter-then-marshal* technique of data customization in SmartTap as **ST1** and the *customized-marshalling* technique as **ST2**. The **No Sampling** data set is obtained when we perform no operation, either at the SmartTap or the DataStager, as a control. Finally, the term **DataStager** refers to the pipeline with execution of C-o-D functions for data customization within the DataStager.

Finally, we show the scalability impact of JITStager on GTC. We evaluate the data filtering and data tagging operations under four different scenarios,

1. **In GTC.** We add the operations to the output function in GTC. This is the least flexible of the options available and requires modification of application itself as well as the associated steps such as revalidation.

2. **SmartTap.** We perform the operation within SmartTap on the compute node. This allows the data filtering to be performed close to the data generation and has the largest impact on the total data size being transferred.

3. **Staging.** We place the specialization operators within the staging area. Although this is a costly operation given the limited computational resources available, compared to the large application cohort, the impact on the application runtime is minimal because the computation is offloaded.

4. **Post I/O.** Finally, we evaluate the time taken to perform the filtering and tagging

operation after the data has been written to disk. This step does not produce any impact on the application runtime, but does have a large latency to completion as well as placing additional stress on the I/O backend.

5.5.1 Understanding the performance of JITStager

Due to the flexibility of the customization engine in JITStager, the policy manager must be aware of the tradeoffs associated with moving computation to the staging area. Data size reduction, time to complete data transfer, and the reduction in resident time on the staging area are the performance benefits of moving computation closer to data generation. However, the additional processing on the compute node can introduce some small overhead in the time required to create the output buffer. The decision to place the customization operation is up to the specialization manager and depends on user specifications.

Figure 13 and Figure 14 show the impact of using a single compute node to perform a data sampling operation. As can be seen, the time required to create the output buffer is increased when we utilize the SmartTap as the sampling engine. This increase in time to marshall is accompanied by a subsequent reduction in data size. One interesting feature of the graphs is the decrease in marshalling time for ST1 as the data retention percentage increases above 50%. This is caused by the increasing effectiveness of the cache due to denser memory references as the data retention percentage increases.

The negative impact on marshalling time is offset by the significant savings within the staging area. To better understand this trade-off, we compare the data residency within the staging area. Data residency is an important metric to consider due to the increase pressure on staging area memory as application node to staging node ratio increases. As can be seen in Figure 15, moving operations to SmartTap results in significant savings in the residency time. Performing data reduction within DataStager also shows I/O time savings (*DataStager (only I/O)* in the Figure 15), but results in greater residency due to the additional computation time required for data reduction. This reduction time is significantly

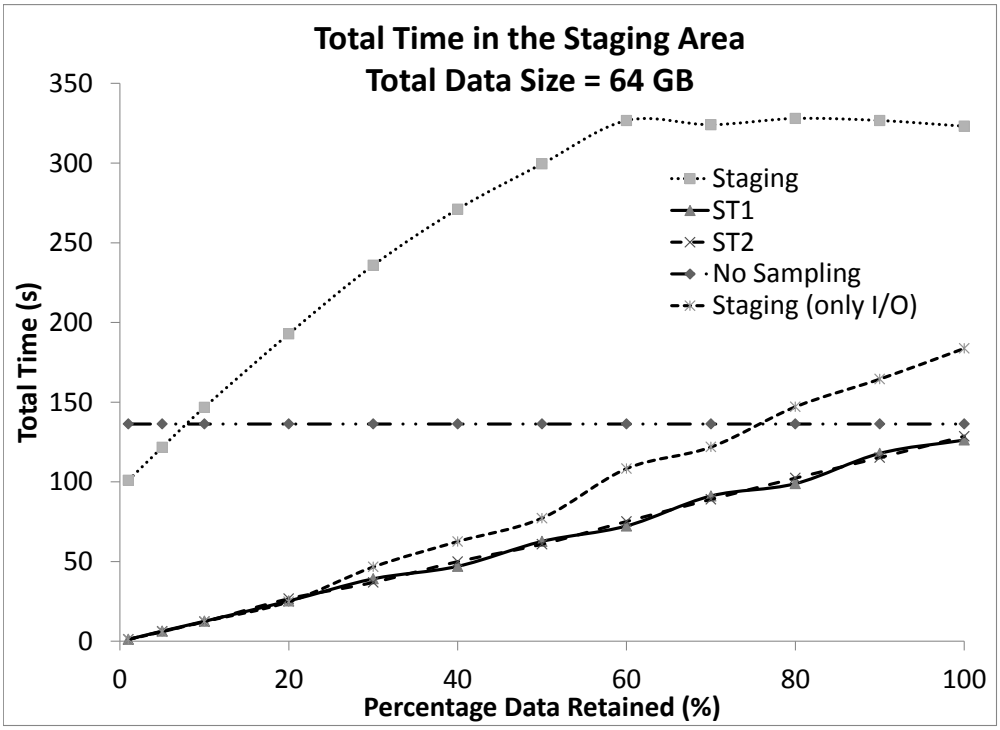


Figure 15: I/O in JITStager: Number of clients = 32, data size = 200 MB.

higher due to the staging area performing the operation on the *entire* data set compared to the SmartTap computing on an *individual* message.

This evaluation shows that the user must have the means to specify policy guidelines that control the mechanism for marshalling, as well as the placement of the sampling code. Additionally, these decisions have to be enforced dynamically to successfully adapt to changing data conditions.

5.5.2 Application scenarios

We evaluate the performance of JITStager with Warp using three of the customizations mentioned in Section 5.2: *Bounding Box*, *Tagged Data Output* and the *Statistical subsample*. We compare these with an extraction operation without any data customization, *Only Data Output*.

The application runs for 1000 iterations with 25 data outputs, every 10 timesteps from

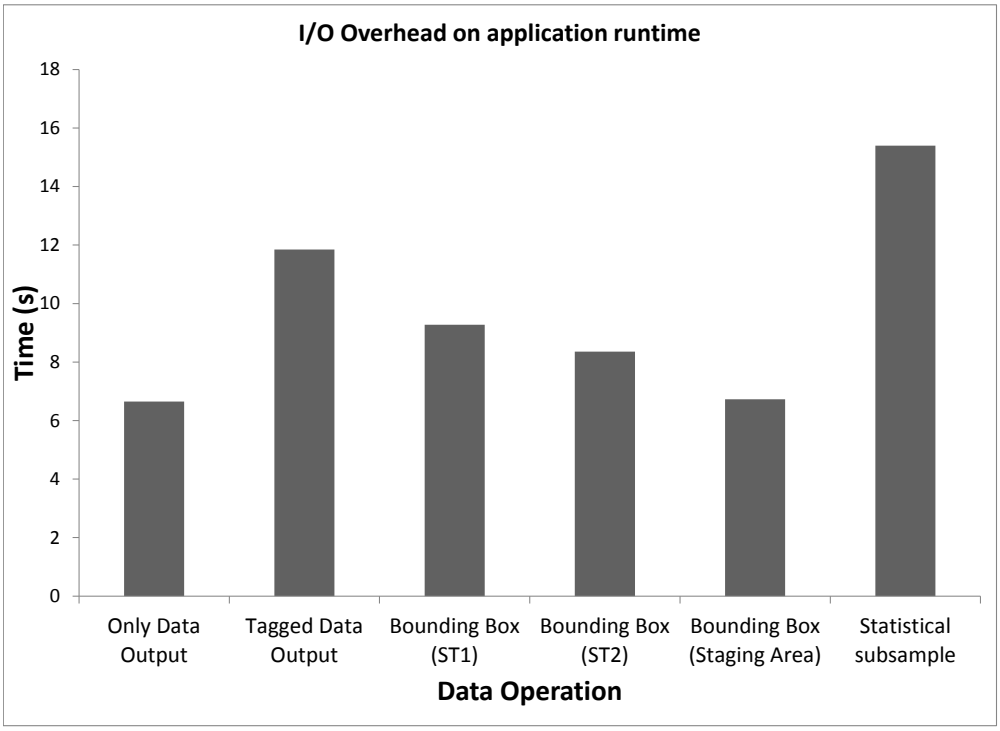


Figure 16: Impact of I/O on the application run of 1000 iterations. Without any I/O, the application runtime was 697s.

0 to 50 and every 50 iterations thereafter. The data output is stored on local disks within the staging area. The experimental platform is a quad-core dual-socket Naehalem cluster with 12 GB of DDR3 ram, using QDR Infiniband as the cluster interconnect. The data size for each output step is 2.4 GB from 32 application processes. The total generated data size is 60 GB for all 25 output steps.

5.5.2.1 Impact on application runtime

Figure 16 shows the overhead on the execution time of the application when utilizing the JITStager I/O transport within ADIOS. The overhead is compared to the NULL transport, a special ADIOS transport that produces no data output. The application run takes $697s$ with the NULL transport. Data output results in an overhead of 0.94%, with the statistical sampling operation creating the largest overhead of $15.4s$. The Tagged Data Output also

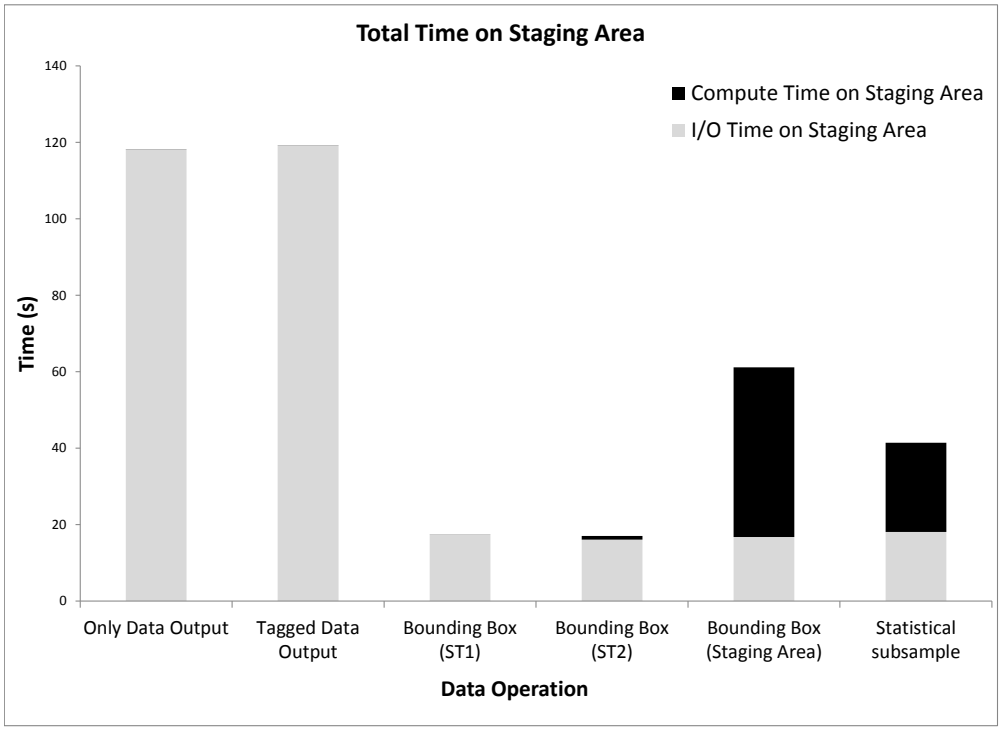


Figure 17: I/O and compute time within the Staging Area.

shows a higher overhead due to the additional computation and the lack of a data reduction. Despite the complexity of these two operations, the overhead is less than 2.5%.

The bounding box operation demonstrates the performance impact of customization placement. Using the C-o-D filter, we see an overhead of only 1.3%, which drops to 1.1% when using the custom marshalling operator. Moving the customization to the staging area results in zero overhead since no additional computation is performed in the SmartTap.

It is important to remember that the overhead from the SmartTap is exaggerated due to the asynchronous data transfer mechanism. With synchronous data output, the I/O time savings through data reduction will dominate the computational cost.

5.5.2.2 Staging area time

In Figure 17, we show the I/O and compute times within the staging area. As shown, the time taken to perform I/O operations within the staging area is substantial compared to the

impact on the application runtime. The total time spent during the 25 output operations is almost $120s$ without any data reduction. When we switch to the bounding box operators, we see the I/O time drop down $17s$. Statistical subsampling produces slightly larger data (see Figure 18) and results in a marginally larger I/O time.

Placing the bounding box operator in DataStager results in similarly reduced I/O time but requires much greater computational time within the staging area. This computational time has no impact on the application runtime, but additional compute time on the staging area reduces the maximum frequency of data output. Moving computation to the staging area eliminates the overhead from SmartTap computation, but has a large impact on time in the staging area. The selection of the placement strategy, therefore, has to be made with client requirements for acceptable application overhead as well I/O overhead requirements for the staging area.

The statistical subsampling operator uses the compute capacity of the staging area to calculate global parameters, resulting in an increased computational time in the staging area. This operation showcases the functionality obtained by combining low overhead annotations in SmartTap, with global feature extraction and functional specialization within DataStager. Computing global features in the application requires global communication operations which impact application scalability.

5.5.2.3 Latency to storage

Storage latency is another important consideration for scientists because it places a limit on how quickly the analytical pipeline can process the extracted data. In Figure 18, we show the reduction in storage latency obtained by using the JITStager customization operations. The latency reduction is highest when the data is reduced and the customization is performed entirely within the SmartTap. Moving the customization to DataStager increases the latency to storage, but the impact is small when compared to writing out the entire uncustomized data set. Statistical sampling shows a higher storage latency due to the time

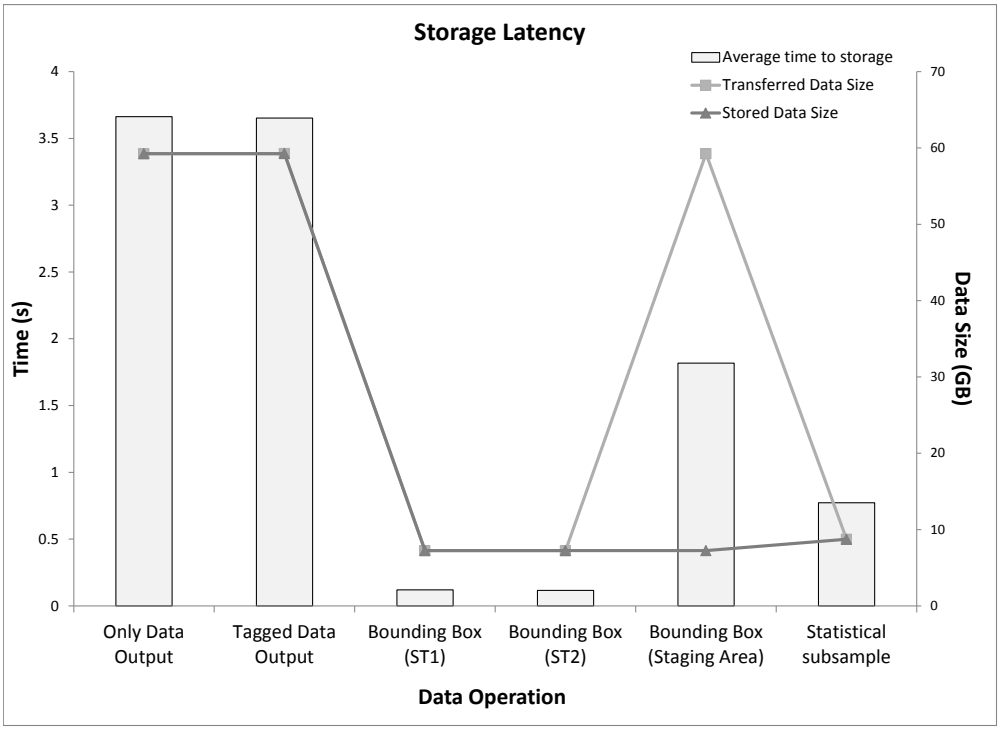


Figure 18: Average latency to storage. With data reductions the data size output to disk is dramatically reduced.

required to compute global features.

5.6 Discussion

The JITStager approach to customizing the I/O pipeline offers both performance and functionality enhancements over previously reported work. The combination of specialization and marshalling on the client side (SmartTap) and of scheduled transfers and transformation on the staging side (DataStager) allows for both a substantial increase in performance flexibility and in extensibility of the I/O pipeline. In contrast to approaches that require fixed, precompiled functions or depend on operating system support, JITStager utilizes on-demand compilation and specialization of the user functions to achieve substantial potential performance gains, such as reducing transfer times from 120s to 17s using user-specified restrictions.

The JITStager, described in this chapter, and the DataTap transport, described in Chapter 4, are the cornerstones of the data services implementation, based on the design detailed in Chapter 3. In the next chapter, we describe how our implementation of data services has addressed some of the challenges described in Chapter 2.

CHAPTER VI

UTILITY OF DATA SERVICES

In this chapter, we describe the use of data services in two of the applications which motivated the development of the data service abstraction. The two applications, CHIMERA and GTC, are commonly used in leadership computing facilities such as the one at Oak Ridge National Laboratory. In particular, this section will concentrate on two major data processing functions that have meaningful implications on real world usage. For CHIMERA, we look at the performance characteristics of extracting data in the native FFS format and converting it to the HDF5 data format more commonly used for analysis tools. Understanding the performance impact of this operation is particularly important, because it allows us to alleviate the concerns of using an intermediate format. In fact we discovered that the performance of data output with FFS format to the staging area and HDF5 conversion within the staging area provided a significant performance benefit. The second common data task we look at is the performance impact of checkpointing for GTC. With large scale simulations, the probability of failure is a significant concern addressed by checkpointing the simulation at regular intervals. Due to the limitations of storage performance, as well as the variability of storage operations, the checkpointing interval is often increased in order to reduce the overall impact on application runtime. By utilizing the data staging and scheduling techniques described previously, we can increase the frequency of checkpointing allowing for less wasted time in case of failures, without greatly impacting the overall application runtime.

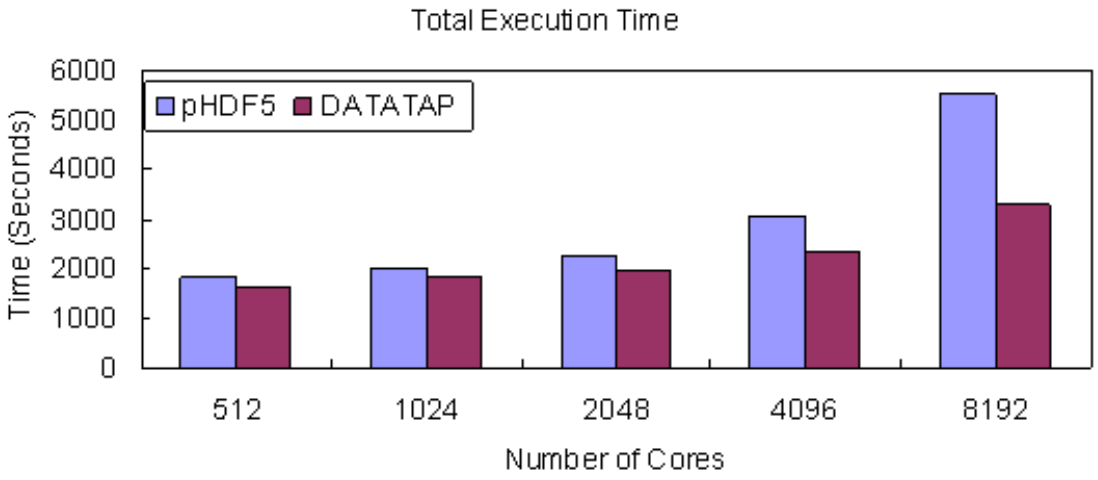


Figure 19: Total Execution Time for CHIMERA.

6.1 Applications

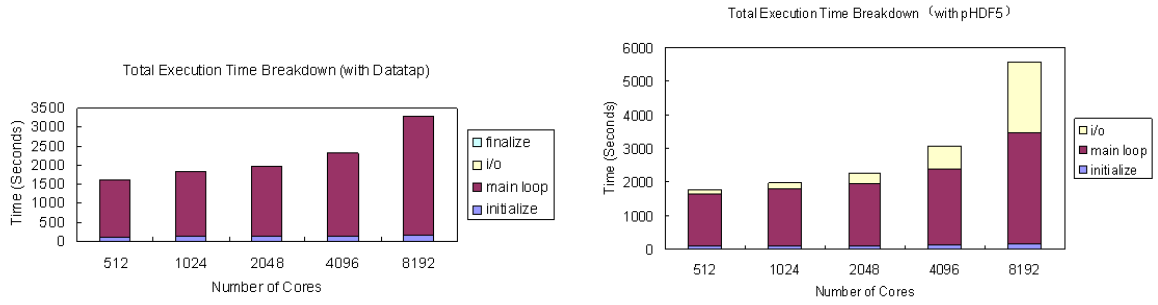
6.1.1 CHIMERA

CHIMERA[52] is a multi-dimensional radiation hydrodynamics code designed to study core-collapse supernovae. We look at the periodic restart output that is used for both checkpointing and post-processing. The restart data consists of 80 scalar and arrays. Global arrays are regularly distributed among a 2-D grid of MPI processes. In our experiments, each MPI process writes out approximately 930KB of data in each I/O phase.CHIMERA uses the ADIOS API [50] for I/O allowing multiple methods to be compared by simply modifying a variable in the configuration file. The data is defined as part of an external XML configuration with both structure and meta information and enabling the use of structured FFS data for output purposes. We have instrumented the application with specific calls to ADIOS in order to provide phase information to the underlying transport method, allowing us to customize the behavior of the data transport.

We evaluate two aspects of the data service for CHIMERA, viz. the extraction of data to the staging area and the conversion of the intermediate FFS format data to the HDF-5 format. Evaluation is performed at application sizes ranging from 512 to 8192 cores. We use five test runs at each of the sizes, with additional compute nodes serving as the data

staging area. In all experiments with the Datatap, we have kept the ratio of compute nodes to staging nodes at 512 to 1. The application runs for 400 iterations and a restart output is produced every 50 application iterations.

The CHIMERA evaluations are performed on the Oak Ridge National Laboratory Cray XT, Jaguar. At the time of these experiments each Jaguar node was a single socket, quad-core AMD Opteron running at 2.1 GHz with 8 GB of memory (2 GB/core). The network interconnect is the Cray Seastar2 with low level access provided through the Portals API, and the compute nodes operating system is Compute Node Linux (CNL).



(a) Total Execution Time Breakdown with Datatap (b) Total Execution Time Breakdown with pHDF5

Figure 20: Comparisons of execution time for CHIMERA.

6.1.2 GTC

Gyrokinetic Toroidal Code (GTC) [71] is a 3-dimensional particle-in-cell code used to study micro-turbulence in magnetic confinement fusion from first principles plasma theory. GTC is highly scalable and our evaluations utilize application sizes from 16k to 112k application cores.

In order to study the largest I/O element, we perform the evaluation on the GTC restart data output only. The GTC restart output is approximately 10% of the overall problem sizes and provides a look at the extremes of I/O performance. Similar to CHIMERA, we use the ADIOS [50] library to perform I/O, providing us with a unique opportunity to study the implications of different methods for data extraction without modifying the application code. The restart output is produced every 10 iterations, and the application runs for a

total of 100 timesteps. The performance of data extraction service is compared to a special ADIOS method, NULL. The NULL method does not perform any data output and provides the base case for application runtime.

As described in [3], we instrument GTC with programmatic hints to inform the data service controller about transitions to a compute phase. GTC evaluations utilize the NCCS Cray XT5 Jaguarpf. Each node is configured with two quad-core AMD Opterons at 2.6 GHz with 16 GB of memory (2GB/core). Like the Cray XT4 used for CHIMERA, the Cray XT5 also uses the SeaStar2+ network interconnect programmed through the Portals API.

As an example of managed data extraction, we use the PA_Con_1 scheduler described in [3]. The scheduler uses phase knowledge from the instrumented application in order to reduce potential interference with intra-application communication and additionally restricts the number of concurrent data transfers. In our past evaluations, we had discovered that the PA_Con_1 scheduler has the least impact on the application runtime.

6.2 Data extraction performance

To examine the impact on CHIMERA performance caused by background data movement, we compare the total execution time of the CHIMERA simulation with Datatap I/O and pHDF5. For Datatap, the visible I/O overhead is the total I/O blocking time in restart dumps plus the time of one-time finalization (during which all compute nodes block waiting for servers to fetch the data).

As shown in Figure 19, data extraction with the Datatap outperforms those with pHDF5. The time breakdown (shown in Figure 20) reveals that the improvement of total execution time is due to reduction of blocking I/O time and that the computation (main loop time) is not affected.

In terms of cost/effectiveness, Table 1 shows that at the scale of 8192 cores, the I/O overhead with pHDF5 is 37.84%. By using 16 additional compute nodes (64 cores in total)

for staging, the I/O overhead is reduced to 0.14%. The additional Datatap servers only cost 64/8192=0.78% additional resources.

Table 1: Visible I/O Overhead

Overhead(%)	512	1024	2048	4096	8192
Datatap	0.0221	0.0248	0.0741	0.142	0.144
pHDF5	7.123	8.925	13.941	22.551	37.837

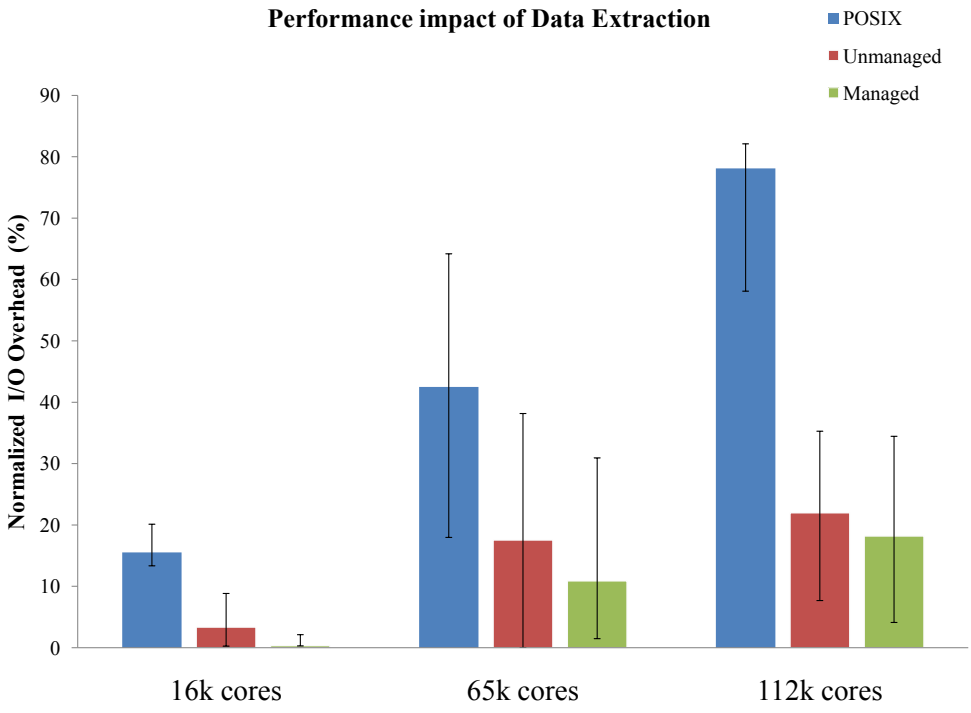


Figure 21: Data Extraction overhead for GTC on the Cray XT5

In Chapter 4, we presented data extraction performance with up to 2k processing cores showing significant performance benefits compared to traditional POSIX output especially when using managed I/O.

In Figure 21, we look at the percentage performance overhead compared to a NULL data transport as we scale from 16,384 to 114,688 processing cores. As the number of processing cores increases from 16k to 65k, the performance overhead increases from 15.5%

to 42.5%. For data extraction using the Datatap using a combination scheduling policy, using the congestion avoidance scheduler (PA) and the concurrency limiting scheduler, we find that the performance overhead was as low as below 1% at 16k cores to only 18.1% at 112k cores.

Figure 22 shows the distribution of blocking times for one representative I/O output over all the nodes in the system.

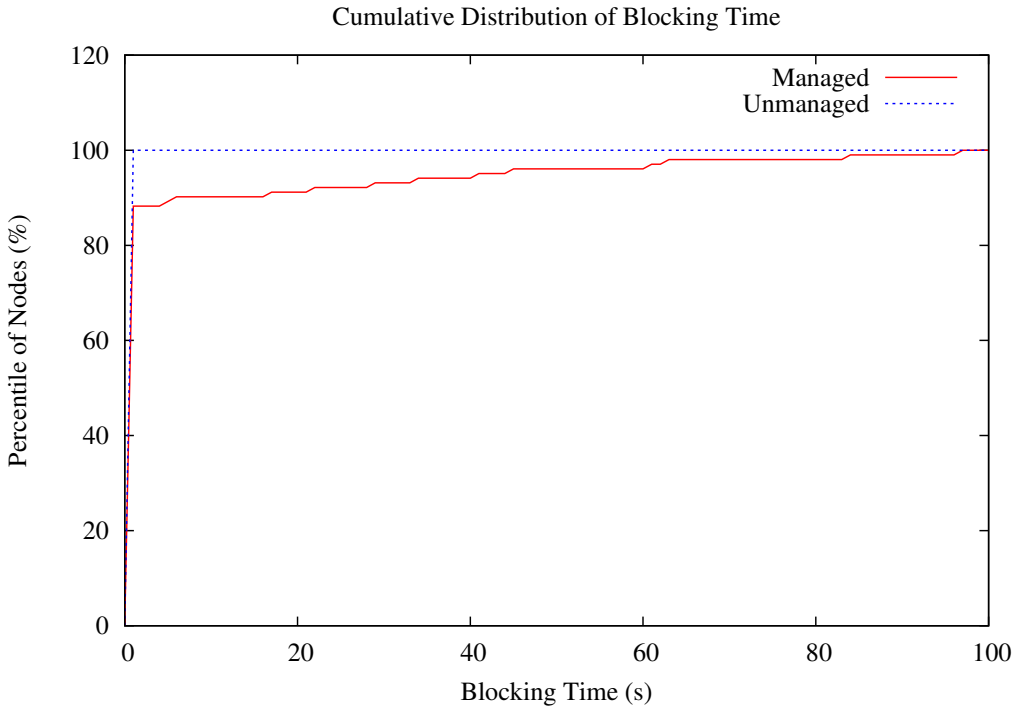


Figure 22: Cumulative distribution of blocking time for a representative I/O phase with 112k processing cores.

Although data extraction to the Datatap server provides a significant performance benefit compared to traditional POSIX I/O, the importance of providing a control infrastructure can be seen from the distribution of blocking time. The unmanaged data stream completes all transfers before the start of the next I/O phase, thus almost 100% of the nodes show a blocking time of less than 2 seconds. However, the managed data transfer limits the time

periods in which data can be moved out of the compute nodes. This results in a small number of nodes (about 10%) taking longer than 10 seconds blocking for the transfer to complete. A tiny fraction, about 1% block waiting for transfer completion for longer than 90 seconds. So, despite the greatly reduced perturbation impact from the managed stream, the overall performance improvement is not as significant. This is an example of a scenario where an independent control plane that optimizes the data movement management for individual application runs can be used for large gains in performance.

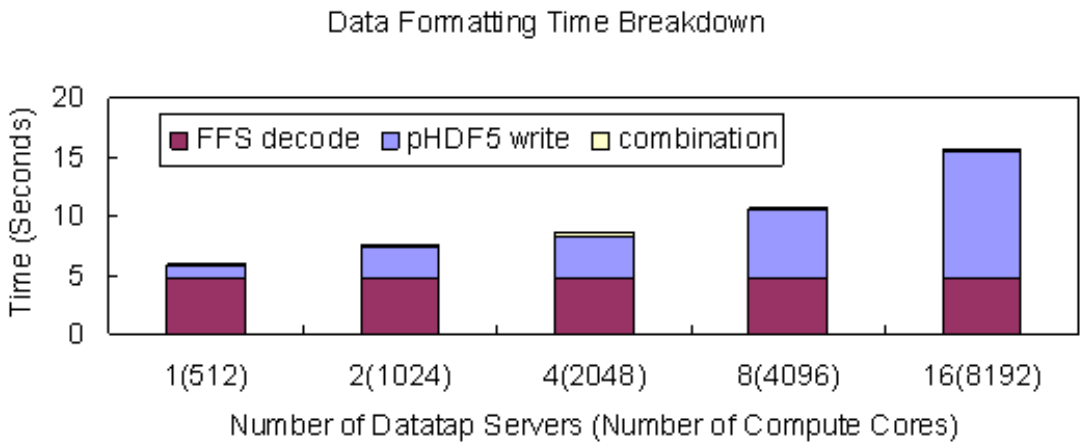


Figure 23: Data Formatting Time.

6.3 Data processing performance

We evaluated the benefits of online data processing using data formatting is for CHIMERA as an example. The data formatting service produces HDF5 formatted data output by utilizing the computational resources of the staging area used for data extraction. Writing data into HDF5 requires first decoding FFS serialized data, combining local arrays into larger chunks, and then writing data to HDF5 file. Figure 23 shows the time breakdown of data formatting. Note that the FFS decoding time shown in Figure 23 is the total time of decoding 512 data chunks. Decoding one message (930KB) takes 0.091 seconds on average. The time of combining data chunks is about 0.25 seconds in total within one dump. These

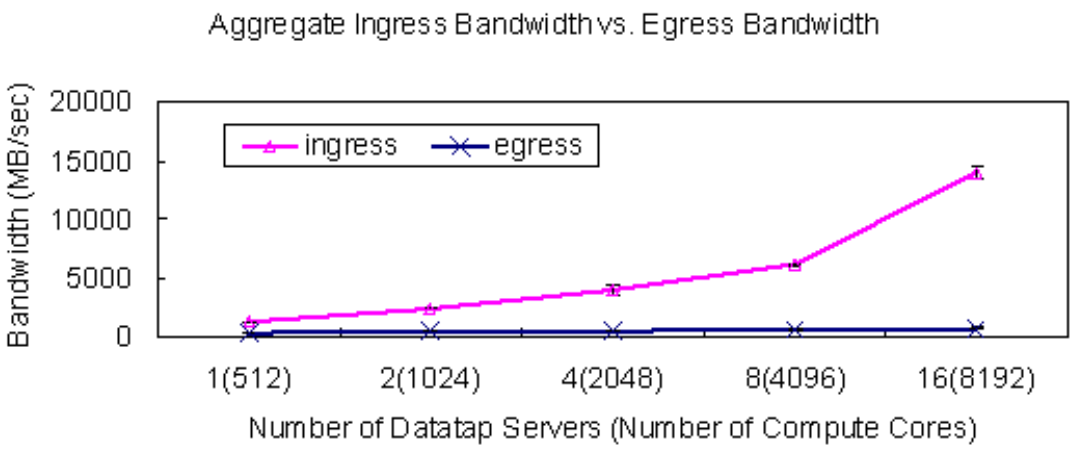


Figure 24: Aggregate ingress and egress bandwidth for HDF5 conversion.

two parts are constant among all tests since the data size is kept unchanged.

However, the time spent in pHDF5 API increases linearly with the total size of the output data. As shown in Figure 24, the egress bandwidth increases only slightly as we increase the number of Datatap servers. This is due to the increase in nodes resulting in an equivalent increase in the coordination and synchronization costs associated with writing a HDF5 file as observed in our previous work [50]. The ingress bandwidth, however, increases more rapidly as we scale the overall application size. The less than linear increase is due to the increased network contention as more nodes are added into the cohort.

Another metric of interest is the total data processing time, which is the time period from getting the first request till finishing writing all data to HDF5 file. The total data processing time at different scales is shown in Figure 25. At the scale of 8192 compute cores, background processing on Datatap servers can be done in less than 20 seconds, while the restart interval is about 400 seconds. This suggests that there is sufficient time for background processing without blocking computation.

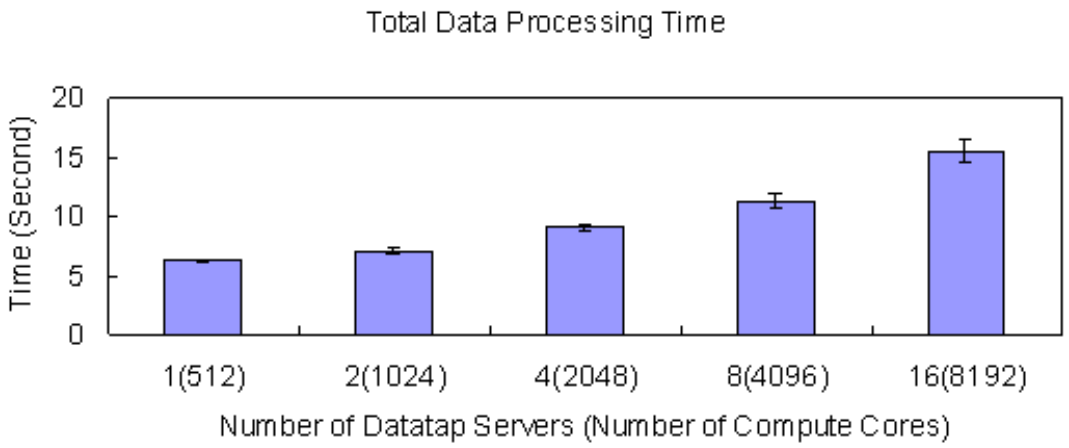


Figure 25: Total Data Processing Time.

6.4 Discussion

Data services have been presented as a useful and usable abstraction for allowing end users to address the problems that come from I/O at extreme scale. By decoupling the management of the I/O transport and storage from the application interface, portability and robustness can be improved for most end users.

In particular, this chapter demonstrates that the separation of extraction, data manipulation, and management of the data fast path can offer significant performance and functionality improvements. Scale measurements over 100k processes show that, as expected, management issues change as the scale of the problem increases. Here we demonstrate that using a combination of management techniques, including asynchronous transport, in-transit filtering of the data, and distributed synchronization, our implementations of data services can perform at or substantially better than the corresponding POSIX-based implementation.

In the next chapter we provide an outlook on the areas where the data services abstraction needs to evolve to address the challenges imposed by the next generation of systems and applications.

CHAPTER VII

LOOKING TO THE FUTURE

The data service abstraction provides a significant improvement over the traditional design of I/O methods for high performance applications. However, there are many directions in which the work in this thesis can be further extended and refined. In particular the ongoing development of the data service architecture requires additional efforts in addressing a wider array of platforms such as the Blue Gene and next generation GPU based HPC systems. Moreover the inclusion of GPUs, and other accelerators, into the next generation of high performance system, raises both new challenges, and also new opportunities for data services research. Additional effort must also be made to investigate the significance of new storage technologies such as NVRAM and SSDs. Finally widescale adoption of new technologies is highly dependent on the development of usable development models and reliable and easy to use deployment techniques.

7.1 Expanding the available platforms

Data Services have been developed given the architecture and platform characteristics of the Cray XT series of high performance systems as well the infiniband based clusters at Georgia Tech. As we move towards multi-petascale and eventually exascale, the architecture will undergo major evolution. In order for data services to gain acceptance for scientific applications, the breadth of the supported platforms must be increased to include the other major architecture commonly used in high performance computing facilities such as Blue Gene class of machines. The increasing penetration of GPUs and other computational accelerators, as well as many-core computational platforms envisioned for exascale architectures must be addressed in extensions to the data service abstraction. The data transport

to the *staging area* must also be extended to support data movement for non uniform memory access (NUMA) architectures as well as heterogeneous cores, both on node and within the staging area. Addressing these issues will require further investigations of the scheduling techniques already demonstrated for asynchronous data movement to the staging area, and also provisioning and scheduling of operations on application nodes utilizing idle or low power cores.

7.2 Utilizing new NVRAM technologies

The current design of data services makes a few strong assumptions about the nature of the data manipulation operations that are performed in the data pipelines. For instance, the data pipeline is assumed to operate as a data stream where the access to output from previous timesteps is not possible. These assumptions are valud for both current and next generation systems, but will not necessarily hold as we scale out systems to exascale. The introduction of non-volatile memory into the system as well the more general availability of fast solid state storage will allow data services to be designed with access to multiple timesteps. This will require the addition of capabilities for temporary storage and retrieval of data objects from these fast random access based persistent stores. Additionally, data services can be extended to combine data objects not just from previous timesteps of a single simulation, but also from previous executions of the simulation. While this adds a large cadre of operations for data services there are significant challenges to be overcome, including discovery of past data, the programming model for the development of these services and the performance considerations from the greater access to storage.

7.3 Programmability and usability

The scope of this thesis did not include an in-depth investigation of the programming model for the development of data services and the control infrastructure required for the deployment of complex I/O pipelines. Ongoing work in the area of parallel global address space

(PGAS) model for development is an important technique to consider for data service development. By describing data in terms of its global characteristics the data services can be easily programmed as parallel services while maintaining scalability. The deployment of data services requires more effort into integration with existing I/O frameworks such as MPI-IO and ADIOS. ADIOS in particular provides significant advantages to utilizing data services by supporting launch time selection of output methods and increasing support in the scientific community. The deployment of dynamic services also requires future work on developing provisioning models for the services, either through user annotations or historical performance data obtained through a monitoring service. The flexibility of the data pipeline can be further exploited by utilizing system level support for fast dynamic allocation of staging resources. Likewise, failure detection and recovery requires further study.

CHAPTER VIII

CONCLUSIONS

The data deluge which has become a significant performance bottleneck for high performance scientific application, requires a paradigm shift in both data output and also data processing. In this thesis we detail our vision of this new paradigm, the data service abstraction. Data services combine low overhead data movement, utilizing the DataStager-DataTap transport, with flexible just in time placement of data processing operations, using the JITStager framework, to address the challenges of these leadership class applications. The use of the scheduling framework in DataStager has been shown to be critical in providing low overhead I/O operations as applications scale to thousand of cores.

The software infrastructure described in this thesis has also been integrated with the ADIOS I/O component framework, allowing the large number of users of ADIOS easy access to this new technology. Implementing the data service abstraction within the framework of ADIOS, provides the technology with a large number of adopters in the scientific computing domain. For example, the particle-in-cell simulation, GTC has addressed many of its I/O challenges by utilizing the techniques described in this thesis. Data services is also being targeted for inclusion in the official release of ADIOS in November 2011, allowing any user of ADIOS to utilize the data services framework, with a minimal level of source code modification.

The use of staging to enable high performance I/O, described in this thesis, is quickly becoming the accepted method of addressing the I/O challenges arising from scaling peak system performance towards exascale. This thesis also presented the first published through evaluation of the performance characteristics of the staging area. The problem of asynchronous I/O causing interference with intra-application collective communication was

initially identified in this thesis. And the proposed solution to this problem, the use of scheduling to avoid contention, is now being considered for inclusion into the development plan for the exascale I/O frameworks.

Data services leverage latent asynchrony present in many scientific workflows, periods of time where the application computes without generating data, and utilizes the novel technique of data staging, described in detail in the thesis, to move data using asynchronous operation to the staging area. The staging area provides a platform for both storing the data temporarily before moving it to persistent storage, and to process the data in order to aid future analysis or visualization operations. This form of preparation for analysis is described as PreDatA, and is one of the use cases for data services and has been shown to have significant benefits for data analysis operations.

The development of meaningful data services has been shown to benefit from the use of self describing binary formats such as FFS. The reflection capability, which forms an integral part of the data description of FFS messages, is used by downstream data services to identify incoming data and operate on it. FFS has also been extended to include in-line sampling to further optimize the marshalling performance by combining the filtering operation with the memory copy to the output buffer.

The use of JITStager extends the locations where data service operations can be executed. In this thesis we have described the use of the dynamic code generation framework, C-on-Demand, to add flexibility to data services in selecting the execution location. JITStager has been shown to provide innovative new functionality, such as statistical filtering of data, without significant impact on application performance. Furthermore, by reducing the data at the source, JITStager has also been shown to reduce the latency to storage and the time to data. The reduction in time to data is especially significant; by providing data faster to the analysis workflow the time taken in creating new scientific knowledge can be reduced. This reduction is coupled with the ability of data services to output only portions of the data set, further reducing the I/O bottlenecks for the scientific workflow.

In summary, data services have been demonstrated to be part of the solution towards scalable data management techniques for the next generation of I/O frameworks. The combination of managed data movement, use of additional resources as the staging area, self describing data formats and dynamic code generation are all combined in providing new functionality to the I/O pipeline. This functionality has been shown to extend beyond simple storage or store-and-forward operations, encompassing a large cadre of scientifically relevant data manipulation operations. This research has already gained extensive adoption in the scientific domain and has opened up new venues for I/O research in the upcoming exascale environments.

REFERENCES

[1] ABBASI, H., WOLF, M., and SCHWAN, K., "LIVE Data Workspace: A Flexible, Dynamic and Extensible Platform for Petascale Applications," *Cluster Computing, 2007. IEEE International*, Sept. 2007.

[2] ABBASI, H., WOLF, M., SCHWAN, K., EISENHAUER, G., and HILTON, A., "Xchange: coupling parallel applications in a dynamic environment," in *CLUSTER '04: Proceedings of the 2004 IEEE International Conference on Cluster Computing*, (Washington, DC, USA), pp. 471–480, IEEE Computer Society, 2004.

[3] ABBASI, H., WOLF, M., EISENHAUER, G., KLASKY, S., SCHWAN, K., and ZHENG, F., "Datastager: scalable data staging services for petascale applications," pp. 39–48, 2009.

[4] ABBASI, H., WOLF, M., EISENHAUER, G., SCHWANN, K., and KLASKY, S., "Just in time: Adding value to the io pipelines of high performance applications with jit-staging," in *The 20th International ACM Symposium on High-Performance Parallel and Distributed Computing*, (San Jose, California), 2011.

[5] AGRAWAL, N., PRABHAKARAN, V., WOBBER, T., DAVIS, J., MANASSE, M., and PANIGRAHY, R., "Design tradeoffs for ssd performance," in *USENIX 2008 Annual Technical Conference on Annual Technical Conference*, pp. 57–70, USENIX Association, 2008.

[6] ALI, N., CARNS, P., ISKRA, K., KIMPE, D., LANG, S., LATHAM, R., ROSS, R., WARD, L., and SADAYAPPAN, P., "Scalable i/o forwarding framework for high-performance computing systems," in *Cluster Computing and Workshops, 2009. CLUSTER'09. IEEE International Conference on*, pp. 1–10, IEEE, 2009.

[7] ALI, N. and LAURIA, M., "Improving the performance of remote i/o using asynchronous primitives," *High Performance Distributed Computing, 2006 15th IEEE International Symposium on*, pp. 218–228, 0-0 0.

[8] ANDERSEN, D., FRANKLIN, J., KAMINSKY, M., PHANISHAYEE, A., TAN, L., and VASUDEVAN, V., "Fawn: A fast array of wimpy nodes," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pp. 1–14, ACM, 2009.

[9] BELL, K., CHIEN, A., and LAURIA, M., "A high-performance cluster storage server," *High Performance Distributed Computing, 2002. HPDC-11 2002. Proceedings. 11th IEEE International Symposium on*, pp. 311–320, 2002.

[10] BEYNON, M., FERREIRA, R., KURC, T. M., SUSSMAN, A., and SALTZ, J. H., "Datacutter: Middleware for filtering very large scientific datasets on archival storage systems," in *IEEE Symposium on Mass Storage Systems*, pp. 119–134, 2000.

[11] BIANCHINI, R., CROVELLA, M., KONTOTHANASSIS, L., and LEBLANC, T., "Alleviating Memory Contention in Matrix Computations on Large-Scale Shared-Memory Multiprocessors," tech. rep., DTIC, 1993.

[12] BORRILL, J., OLIKER, L., SHALF, J., and SHAN, H., "Investigation Of Leading HPC I/O Performance Using A Scientific-Application Derived Benchmark," in *Proceedings of the 2007 Conference on SuperComputing, SC07*, 2007.

[13] BRIGHTWELL, R., HUDSON, T., RIESEN, R., and MACCABE, A. B., "The Portals 3.0 message passing interface," Technical report SAND99-2959, Sandia National Laboratories, December 1999.

[14] BRIGHTWELL, R., LAWRY, B., MACCABE, A. B., and RIESEN, R., "Portals 3.0: Protocol building blocks for low overhead communication," in *IPDPS '02: Proceedings of the 16th International Parallel and Distributed Processing Symposium*, (Washington, DC, USA), p. 268, IEEE Computer Society, 2002.

[15] BUSTAMANTE, F. E., EISENHAUER, G., SCHWAN, K., and WIDENER, P., "Efficient wire formats for high performance computing," in *Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, p. 39, IEEE Computer Society, 2000.

[16] CAI, Z., EISENHAUER, G., HE, Q., KUMAR, V., SCHWAN, K., and WOLF, M., "Iq-services: network-aware middleware for interactive large-data applications," in *MGC '04: Proceedings of the 2nd workshop on Middleware for grid computing*, (New York, NY, USA), pp. 11–16, ACM Press, 2004.

[17] CARNS, P. H., LIGON III, W. B., ROSS, R. B., and THAKUR, R., "PVFS: A parallel file system for linux clusters," in *Proceedings of the 4th Annual Linux Showcase and Conference*, (Atlanta, GA), pp. 317–327, USENIX Association, 2000.

[18] "Lustre: A scalable, high-performance file system." Cluster File Systems Inc. white paper, version 1.0, November 2002. http://www.lustre.org/docs/whitepaper.pdf.

[19] COMMITTEE, T. M.-I., "Mpi-io: A parallel file i/o interface for mpi," April 1996. Version 0.5. MPI-2 standard is at http://www.mcs.anl.gov/mpi.

[20] DANDAMUDI, S., "Reducing hot-spot contention in shared-memory multiprocessor systems," *Concurrency, IEEE [see also IEEE Parallel & Distributed Technology]*, vol. 7, no. 1, pp. 48–59, Jan-Mar 1999.

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

[22] DING, C., DWARKADAS, S., HUANG, M., SHEN, K., and CARTER, J., "Program phase detection and exploitation," *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, pp. 8 pp.–, 25-29 April 2006.

[23] DOCAN, C., PARASHAR, M., and KLASKY, S., "High speed asynchronous data transfers on the cray xt3," in *Cray User Group Conference*, 2007.

[24] DOCAN, C., PARASHAR, M., CUMMINGS, J., PODHORSZKI, N., and KLASKY, S., "Experiments with Memory-to-Memory Coupling for End-to-End Fusion Simulation Workflows.," Tech. Rep. TR-104, Center for Autonomic Computing (CAC), Rutgers University, Piscataway, NJ, USA, July 2009.

[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 2010)*, June 2010.

[26] EISENHAUER, G., "The evpath library." `http://www.cc.gatech.edu/systems/projects/EVPath`.

[27] EISENHAUER, G., "Portable binary input/output." `http://www.cc.gatech.edu/systems/projects/PBIO`.

[28] EISENHAUER, G., "The ECho Event Delivery System," Tech. Rep. GIT-CC-99-08, Georgia Tech, Aug 1999.

[29] EISENHAUER, G., BUSTAMENTE, F., and SCHWAN, K., "Event services for high performance computing," in *Proceedings of High Performance Distributed Computing (HPDC-2000)*, 2000.

[30] EKANAYAKE, J., PALLICKARA, S., and FOX, G., "Mapreduce for data intensive scientific analyses," in *ESCIENCE '08: Proceedings of the 2008 Fourth IEEE International Conference on eScience*, (Washington, DC, USA), pp. 277–284, IEEE Computer Society, 2008.

[31] FOSTER, I., VOCKLER, J., WILDE, M., and ZHAO, Y., "Chimera: A virtual data system for representing, querying, and automating data derivation," in *Scientific and Statistical Database Management, 2002. Proceedings. 14th International Conference on*, pp. 37–46, 2002.

[32] FREITAS, R. and WILCKE, W., "Storage-class memory: The next storage system technology," *IBM Journal of Research and Development*, vol. 52, no. 4.5, pp. 439–447, 2008.

[33] GANEV, I., EISENHAUER, G., and SCHWAN, K., "Kernel Plugins: When a VM is Too Much," in *Proceedings of the 3rd Virtual Machine Research and Technology Symposium (VM'04)*, (San Jose, CA), May 2004.

[34] GOLESTANI, S., "A stop-and-go queueing framework for congestion management," in *SIGCOMM'90 Symposium*, pp. 8–18, ACM, September 1990.

[35] INTEL, "Virtualization technology (vt)." http://www.intel.com/technology/computing/vptech/.

[36] ISKRA, K., ROMEIN, J., YOSHII, K., and BECKMAN, P., "ZOID: I/O-forwarding infrastructure for petascale architectures," in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pp. 153–162, ACM, 2008.

[37] JAIN, R., RAMAKRISHNAN, K. K., and CHIU, D. M., "Congestion avoidance in computer networks with a connectionless network layer," Tech. Rep. DEC-TR-506, Digital Equipment Corporation, MA, Aug. 1987.

[38] JIANG, N., QUIROZ, A., SCHMIDT, C., and PARASHAR, M., "Meteor: a middleware infrastructure for content-based decoupled interactions in pervasive grid environments," *Concurrency and Computation: Practice and Experience*, vol. 20, no. 12, pp. 1455–1484, 2008.

[39] JONES, G. S., "Fusion gets faster," July 2009.

[40] KENG LIAO, W. and CHOUDHARY, A. N., "Dynamically adapting file domain partitioning methods for collective i/o based on underlying parallel file system locking protocols," in *Proceedings of the ACM/IEEE Conference on High Performance Computing, SC 2008, November 15-21, 2008, Austin, Texas, USA*, p. 3, IEEE/ACM, 2008.

[41] KLASKY, S., ETHIER, S., LIN, Z., MARTINS, K., MCCUNE, D., and SAMTANEY, R., "Grid -based parallel data streaming implemented for the gyrokinetic toroidal code," in *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, (Washington, DC, USA), p. 24, IEEE Computer Society, 2003.

[42] KOTZ, D., "Disk-directed I/O for MIMD multiprocessors," *ACM Transactions on Computer Systems*, vol. 15, pp. 41–47, February 1997.

[43] KUBIATOWICZ, J., BINDEL, D., CHEN, Y., CZERWINSKI, S., EATON, P., GEELS, D., GUMMADI, R., RHEA, S., WEATHERSPOON, H., WELLS, C., and OTHERS, "Oceanstore: An architecture for global-scale persistent storage," *ACM SIGARCH Computer Architecture News*, vol. 28, no. 5, pp. 190–201, 2000.

[44] KUMAR, V., CAI, Z., COOPER, B. F., EISENHAUER, G., SCHWAN, K., MANSOUR, M., SESHASAYEE, B., and WIDENER, P., "Implementing diverse messaging models with self-managing properties using iflow," in *3rd IEEE International Conference on Autonomic Computing (ICAC)*, 2006.

[45] KUTARE, M., EISENHAUER, G., WANG, C., SCHWAN, K., TALWAR, V., and WOLF, M., "Monalytics: Online monitoring and analytics for managing large scale data centers," in *The 7th IEEE/ACM International Conference on Autonomic Computing and Communications*, June 2010.

[46] LATHAM, R., MILLER, N., ROSS, R., and CARNS, P., "A next-generation parallel file system for linux clusters," *LinuxWorld*, vol. 2, January 2004.

[47] LEE, J., ROSS, R., ATCHLEY, S., BECK, M., and THAKUR, R., "Mpi-io/l: efficient remote i/o for mpi-io via logistical networking," in *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, pp. 10–pp, IEEE, 2006.

[48] LOFSTEAD, J., SCHWAN, K., KLASKY, S., PODHORSZKI, N., and JIN, C., "Flexible io and integration for scientific codes through the adaptable io system (adios)," in *Challenges of Large Applications in Distributed Environments (CLADE)*, 2008.

[49] LOFSTEAD, J., ZHENG, F., KLASKY, S., and SCHWAN, K., "Adaptable, metadata rich io methods for portable high performance io," in *In IPDPS'09, May 25-29, Rome, Italy*, 2009.

[50] LOFSTEAD, J. F., KLASKY, S., SCHWAN, K., PODHORSZKI, N., and JIN, C., "Flexible io and integration for scientific codes through the adaptable io system (adios)," in *CLADE '08: Proceedings of the 6th international workshop on Challenges of large applications in distributed environments*, (New York, NY, USA), pp. 15–24, ACM, 2008.

[51] LUDSCHER, 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, no. 10, pp. 1039–1065, 2006.

[52] MESSER, O. E. B., BRUENN, S. W., BLONDIN, J. M., HIX, W. R., MEZZACAPPA, A., and DIRK, C. J., "Petascale supernova simulation with CHIMERA," *Journal of Physics Conference Series*, vol. 78, pp. 012049–+, July 2007.

[53] MILLER, B. P., CALLAGHAN, M. D., CARGILLE, J. M., HOLLINGSWORTH, J. K., IRVIN, R. B., KARAVANIC, K. L., KUNCHITHAPADAM, K., and NEWHALL, T., "The paradyn parallel performance measurement tool," *IEEE Computer*, vol. 28, no. 11, pp. 37–46, 1995.

[54] MILLER, E. L. and KATZ, R. H., "Input/output behavior of supercomputing applications," in *Supercomputing '91: Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, (New York, NY, USA), pp. 567–576, ACM, 1991.

[55] NAGLE, D., SERENYI, D., and MATTHEWS, A., "The panasas activescale storage cluster: Delivering scalable high bandwidth storage," in *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, p. 53, IEEE Computer Society, 2004.

[56] NARAYANAN, D., THERESKA, E., DONNELLY, A., ELNIKETY, S., and ROWSTRON, A., "Migrating server storage to ssds: analysis of tradeoffs," in *Proceedings of the 4th ACM European conference on Computer systems*, pp. 145–158, ACM, 2009.

[57] NISAR, A., KENG LIAO, W., and CHOUDHARY, A. N., "Scaling parallel i/o performance through i/o delegate and caching system," in *Proceedings of the ACM/IEEE Conference on High Performance Computing, SC 2008, November 15-21, 2008, Austin, Texas, USA*, p. 9, 2008.

[58] OLDFIELD, R. A., MACCABE, A. B., ARUNAGIRI, S., KORDENBROCK, T., SEN, R. R., WARD, L., and WIDENER, P., "Lightweight I/O for Scientific Applications," in *Proc. 2006 IEEE Conference on Cluster Computing*, (Barcelona, Spain), September 2006.

[59] OLDFIELD, R. A., WIDENER, P., MACCABE, A. B., WARD, L., and KORDENBROCK, T., "Efficient Data Movement for Lightweight I/O," in *Proc. 2006 Workshop on high-performance I/O techniques and deployment of Very-Large Scale I/ O Systems (HiPerI/O 2006)*, (Barcelona, Spain), September 2006.

[60] OLIKER, L., CARTER, J., MICHAEL WEHNER, CANNING, A., ETHIER, S., MIRIN, A., BALA, G., PARKS, D., PATRICK WORLEY SHIGEMUNE KITAWAKI, and TSUDA, Y., "Leading computational methods on scalar and vector hec platforms," in *Proceedings of SuperComputing 2005*, 2005.

[61] PATRICK, C. M., SON, S., and KANDEMIR, M., "Comparative evaluation of overlap strategies with study of i/o overlap in mpi-io," *SIGOPS Oper. Syst. Rev.*, vol. 42, no. 6, pp. 43–49, 2008.

[62] PLIMPTON, S., "Fast parallel algorithms for short-range molecular dynamics," *Journal of Computational Physics*, vol. 117, no. 1, pp. 1–19, 1995. `http://lammps.sandia.gov/index.html`.

[63] POLTE, M., SIMSA, J., TANTISIRIROJ, W., and GIBSON, G., "Fast log-based concurrent writing of checkpoints," in *In Proceedings of Petascale Data Storage Workshop 2008 at Supercomputing 2008*, 2008.

[64] PRABHAT, WU, J. K., CHUO, J., HOWISON, M., BETHEL, E. W., KOZIOL, Q., PALMER, B., and SCHUCHARDT, K., "Exahdf5 an i/o platform for exascale data models, analysis and performance," March 2011.

[65] ROTH, P., ARNOLD, D., and MILLER, B., "Mrnet: A software-based multicast/reduction network for scalable tools," in *Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, p. 21, IEEE Computer Society, 2003.

[66] SCHMUCK, F. and HASKIN, R., "GPFS: A Shared-Disk File System for Large Computing Clusters," *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, 2002.

[67] SINHA, S. and PARASHAR, M., "Adaptive system sensitive partitioning of amr applications on heterogeneous clusters," *Cluster Computing*, vol. 5, no. 4, pp. 343–352, 2002.

[68] SOUNDARARAJAN, G., PRABHAKARAN, V., BALAKRISHNAN, M., and WOBBER, T., "Extending ssd lifetimes with disk-based write caches," in *Proceedings of the 8th USENIX conference on File and storage technologies*, pp. 8–8, USENIX Association, 2010.

[69] STONE, N., BALOG, D., GILL, B., JOHAN-SON, B., MARSTELLER, J., NOWOCZYNSKI, P., PORTER, D., REDDY, R., SCOTT, J., SIMMEL, D., and OTHERS, "PDIO: High-performance remote file I/O for Portals enabled compute nodes," *Proceedings of the 2006 Conference on Parallel and Distributed Processing Techniques and Applications, Las Vegas, NV, June*, 2006.

[70] TITZER, B. L., WÜRTHINGER, T., SIMON, D., and CINTRA, M., "Improving compiler-runtime separation with xir," in *VEE '10: Proceedings of the 6th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, (New York, NY, USA), pp. 39–50, ACM, 2010.

[71] WANG, W., LIN, Z., TANG, W., LEE, W., ETHIER, S., LEWANDOWSKI, J., REWOLDT, G., HAHM, T., and MANICKAM, J., "Gyro-kinetic simulation of global turbulent transport properties in tokamak experiments," *Physics of Plasmas*, vol. 13, p. 092505, 2006.

[72] WIDENER, P. M., WOLF, M., ABBASI, H., BARRICK, M., LOFSTEAD, J., PULLIKOTTIL, J., EISENHAUER, G., GAVRILOVSKA, A., KLASKY, S., OLDFIELD, R., BRIDGES, P. G., MACCABE, A. B., and SCHWAN, K., "Structured streams: Data services for petascale science environments," Tech. Rep. TR-CS-2007-17, University of New Mexico, Albuquerque, NM, November 2007.

[73] WOLF, M., ABBASI, H., COLLINS, B., SPAIN, D., and SCHWAN, K., "Service augmentation for high end interactive data services," in *IEEE International Conference on Cluster Computing (Cluster 2005)*, September 2005.

[74] YU, W., VETTER, J., and ORAL, H., "Performance characterization and optimization of parallel i/o on the cray xt," in *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pp. 1–11, IEEE, 2008.

[75] ZHENG, F., ABBASI, H., DOCAN, C., LOFSTEAD, J., LIU, Q., KLASKY, S., PARASHAR, M., PODHORSZKI, N., SCHWAN, K., and WOLF, M., "Predata - preparatory data analytics on peta-scale machines," in *Proceedings of the 24th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2010)*, April 2010.