SAND2015-0207C

I/O Containers: Management Abstractions for Large-Scale Science Workflows

Jai Dayal
College of Computing
Georgia Institute of Technology
Atlanta, GA
jdayal3@gatech.edu

Jay Lofstead
Computer Science Research Institute
Sandia National Laboratories
Albuquerque, NM
gflofst@sandia.gov

Greg Eisenhauer
College of Computing
Georgia Institute of Technology
Atlanta, GA
eisen@cc.gatech.edu

Karsten Schwan
College of Computing
Georgia Institute of Technology
Atlanta, GA
schwan@cc.gatech.edu

Matthew Wolf
College of Computing
Georgia Institute of Technology
Atlanta, GA
mwolf@cc.gatech.edu

Hasan Abbasi
Oak Ridge National Laboratories
Oak Ridge, TN
habbasi@ornl.gov

Scott Klasky
Oak Ridge National Laboratories
Oak Ridge, TN
klasky@ornl.gov

ABSTRACT

The path towards exascale has given rise to a new model of scientific inquiry where concurrently with the running simulation, online analytics workflows operate on the data it produces. While speeding up the scientific discovery process by providing rapid insights into the simulated science phenomena, a challenge for online analytics is to respond to workflow behavior dynamics caused by changing simulation outputs and by unforeseen events on the underlying hardware/software platforms.

This paper presents a set of run-time abstractions for online workflow management, realized by embedding workflow components into "I/O Containers" that monitor component behavior and enable responses to runtime changes in their resource usage and in the platform's resource availability. Management actions concern individual components and the end-to-end properties of entire workflows through a hierarchical management infrastructure.

For high end simulations running on a leadership machine, experimental evaluations show I/O containers can invoke efficient management operations responding to runtime dynamics at different analytics workflow granularities.

General Terms

High-performance Computing

Keywords

Data Staging;Data Management;Analytics

1. INTRODUCTION

On current generation petascale platforms, scientific applications like the GTC [22] fusion and S3D [17] combustion simulations are already generating terabytes of data every few minutes. The desire to scale the I/O and the analytics and visualization codes operating on such data to exascale levels has caused researchers to devise new online methods for managing the large data volumes without overwhelming the parallel file systems attached to these machines. These methods include running analytics concurrently along side simulations – "in-situ" [41, 7] – and in I/O staging areas – "in-transit" [4, 14, 18] – on the high end machine and/or extending to auxiliary analytics clusters.

Beyond addressing performance challenges, online analytics offer science users new functionality for better understanding the scientific simulations being run. This includes (i) continuously ascertaining simulation validity, permitting it to be terminated or corrected without undue waste of machine resources [23], (ii) gaining rapid insights into the scientific processes being simulated (online visualization), or even (iii) enabling methods for application steering. The result of these developments, however, is that at exascale, projections suggest that high-end codes will no longer be structured as a single, large, synchronous application, but rather as a set of components running concurrently with the simulation that ingest and operate on simulation output data. This combination of analytics components deployed into the simulation's I/O path is termed an *I/O pipeline*.

In contrast to the long-running and often well-tuned simulations, there are considerable variations in the analytics codes present in I/O pipelines. They differ in their maturity, degrees of parallelism, execution models, data characteristics, resilience capabilities, and others. They can also exhibit substantially dynamic execution behavior in part due to their data-dependent functionality. For example, an analytics code's runtime is determined by the number of features found in the output data it analyzes. I/O pipelines, therefore, can experience dynamic changes in resource consumption and requirements making their initial resource allocations inappropriate and/or requiring adjustments in how analytics operate. For example, reducing a component's precision or similar measures can rebalance resource usage. It is also possible that some analytics may simply be too expensive to run online for certain kinds of data outputs due to structural issues like insufficient parallelism or because they require further tuning for coping with such outputs. In fact, even a single slow component in an I/O pipeline can inhibit the entire pipeline's performance, as amply demonstrated in past research for both HPC [14, 39] systems and for the multi-tier services run by web companies [33, 21].

Failure to react to online changes in I/O pipeline behavior can lead to severe consequences. Unduly slow analytics pipelines can cause data loss or worse, stall high end simulations by causing them to block on their output actions. Offline tuning driven by continuous performance profiling is one way to address the problem, but its use requires stable I/O pipelines. This also prevents end users from experimenting with interesting new analytics or visualizations for understanding simulation behavior. In response, both in the web domain and in high performance computing, developers are increasingly looking toward online solutions for controlling analytics behavior and resource consumption [27, 12]. The aim of such methods is to manage the diverse sets of codes and resources contained in an I/O pipeline so as to ensure the efficient, high performance, and correct execution of entire I/O pipelines, both for their individual and potentially parallel components and for their end-to-end properties.

This paper describes the *I/O container* approach, depicted in Fig. 1, to managing dynamic I/O and analytics pipelines on high end machines. I/O containers permit developers to embed their analytics functions into a componentized, dynamically managed execution and messaging framework. Such components can be compiled and deployed separately, each in their own container, have well defined inputs and outputs [26], can be parallel (MPI or threads), and may exhibit inter-component dependencies. Entire I/O pipelines or workflows can be constructed by chaining containers along their I/O paths.

I/O containers offer:

1. *controlled resource usage:* a container provides and manages resources for the analytics component mapped to it;

2. *per-component management:* a container offers to its component an actively managed execution environment and allows components to perform customized implementations of management operations to ensure that their own local properties and requirements are not violated;

3. *metric-driven operation:* the container runtime can also enforce goals driven by metrics of interest to end users, such as priorities or performance requirements; containers are thus continually monitored to provide managers with the information needed to make management decisions.

An additional property of containers is their fault-resilient management, through transactional techniques that guarantee that the control and management actions taken by container software do not place applications or analytics components into inconsistent states [25], for example, not making use of a resource until a different container has fully relinquished it. Such requirements become important as online science workflows scale geographically [6] as network partitions or data center outages can render parts of the workflow inoperable.

Using I/O containers permits end users to focus on analytics functionality and algorithmic correctness rather than being overly concerned with scaling individual analytics components and/or their careful resource allocations. Instead, with containers, users can specify customized SLAs and management actions to be performed to ensure that certain desired SLA properties are met with container managers responsible for maintaining per component and the resulting globally, i.e., end-to-end, desired SLAs. A typical management division is one in which a global manager is responsible for maintaining an entire workflow's SLA. It will achieve this by re-organizing its containers and container-level managers perform actions specific to individual components, e.g., by changing a component's degree of internal parallelism. A concrete demonstration of such functionality in this paper is one in which multiple I/O containers segment the single, common staging area used to execute online analytics for a scientific simulation. One such container may run a data visualization with, for example, VTK [28], while another may run analytics using ParaView [10]. A dynamic requirement for additional resources to run ParaViews's analytics can be met by 'stealing' resources from the visualization container, if it does not need them, or by using spare staging resources, if available.

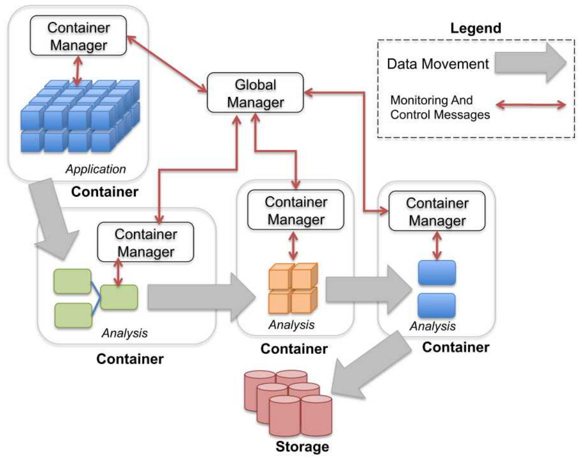


Figure 1: High-level view of I/O Containers framework.

I/O containers with their well-defined component interfaces and a programmatic management API expose to end users management primitives for specifying their SLAs, indicating the appropriate actions to take when certain conditions are detected. Condition detection operates at runtime, for each container and delivered to the management hierarchy via continuous online monitoring of individual containers concerning their current behavior and resource usage.

The implementation of I/O containers evaluated in this paper is based on the EVPath event messaging system and the Flexpath staging solution [15, 11], able to run on cluster machines and on high end supercomputers like ORNL's Titan. Using two high end applications, the LAMMPS [30] molecular dynamics and the GTS [34] fusion simulations, along with different sets of analytics pipelines (SmartPointer [36] and an FFT code, respectively), we evaluate I/O containers with SLAs that include: (1) a global performance-driven SLA that implements "elastic containers" to recover from detected bottlenecks (throughput degredation) in the I/O pipeline; (2) a container-level data-centric policy that executes specific SmartPointer analysis routines in response to certain features being detected in the output data it is processing; and (3) a set of fault recovery policies to recover from an unexpected departure of a component when executing analysis codes on an end-user device (e.g, a laptop).

Experimental evaluations show that the use of such active container-based management can: (1) respond to runtime dynamics at different levels of the stack; (2) create and enforce SLAs at multiple granularities of an I/O pipeline; and (3) operate at large scales with low overheads.

I/O containers constitute new functionality in the domain of scientific data management. Current current I/O staging technologies do not yet offer support for dynamically managing the end-to-end properties of tightly coupled analytics running with high end codes. In earlier work on data staging, for instance, statically profiled analysis routines are run in configurations sized to be resource-rich for worst case data volumes and processing needs [40]. Similarly, our recent work on 'in-situ' analytics for supercomputer simulations [41] schedules and manages only the analytics actions taking place on individual compute nodes not being concerned about the I/O pipeline end-to-end properties originating at such nodes.

Previous work on datacenter management and for 'big data' systems uses techniques like elasticity and replication, to provide scalability and fault tolerance [19, 27, 38, 2]. Such work motivates some of our work, but its software realizations are not suitable for the HPC domain and its methods do not directly address the end-to-end behaviors of the parallel analytics workflows managed by I/O containers. Specifically, with the I/O container model and its information infrastructure, we can realize the diverse management semantics needed for such workflows and science end users, expressed with SLAs, and driving management actions that implement the limited types of elasticity permitted by the HPC machine, the degree of reactivity needed for effective workflow use, and the desired end-to-end behaviors, such as throughput or latency. Initial results [12] demonstrate some of these properties, but the work presented in this paper (i) extends the container workflow model and management constructs, (ii) explores a wider variety of use cases, including an understanding of how state and metadata are managed (i.e., quality of data and fault recovery), (iii) describes how SLAs are defined and how management policies are constructed to enforce them, and (iv) extends the concepts to workflows that span multiple machines by leveraging the Flexpath [11] staging solution operating across a variety of interconnects (our earlier solution implemented with the DataTap [4] staging solution operating on the Cray Portals API [8] operated only on the high end machine).

The remainder of this paper is organized as follows. Section 2 presents the I/O container concept, desired features, and design and implementation. The performance evaluation in Section 4 demonstrates the strength of the approach. Section 5 discusses relevant research related to the containers concept. Section 6 concludes the paper and discusses future work.

2. I/O CONTAINERS

I/O containers are run-time abstractions that allow in-transit data processing actions to be embedded into a dynamically managed execution environment. Each single container manages executables that carry out analytics tasks on the data they ingest. More complex structures, like entire I/O pipelines, are supported by chains of containers supervised by a global manager interacting with per-container managers. Such distributed management is supported with a flexible monitoring and control infrastructure gathering needed information and then issuing appropriate control operations. In all such cases, analytics components are run on the machine resources allocated to the container infrastructure by the user and controlled by globally and potentially container-specific management and scheduling. This multi-level management scheme can maintain both container-level and global (i.e., across all containers) properties.

Fig. 1 depicts a conceptual model of I/O containers.

2.1 Assumptions and Desired Properties

The I/O container approach rests on assumptions that hold true for many large-scale scientific applications and their associated online analytics workflows. These assumptions do not always match those found in enterprise or 'big data' frameworks like [37, 2, 1] in terms of their data characteristics, execution models, and degrees of parallelism.

- **Functional Dependencies.** Analytics codes expect to ingest data matching specific formats and layouts, where analytics functions may need to transform data to meet algorithmic correctness and/or to export an analysis function's discoveries into the data itself. Given these dependencies in the data-plane, functions in an analytics workflow may not have the ability to operate out of order; one function's inputs often requires another function's outputs.

- **Heterogeneous Codes.** Analytics codes can be heterogeneous in terms of their parallelism, execution models, fault tolerance, and scaling characteristics. This results in substantial variations across different I/O containers and the components they manage.

- **Stringent Resource Constraints.** Resources are not free, since the bulk of the resources are typically assigned to the simulations being run, whereas analysis codes are given 'spare' resources, i.e., spare CPU cycles on simulation nodes [41, 7], reserved staging nodes [4, 14], or those on smaller, auxiliary clusters perhaps in different physical locations. Analytics workflows, therefore, must operate with these limited resources, without interfering with the simulations and their output actions.

Given these assumptions and the set of challenges and application characteristics outlined above, I/O container-based workflow management must meet the following design goals. First, given the large variety of characteristics of analytics codes and the dynamics they experience at runtime, it is impractical for a single manager to understand all analytics in some composed I/O workflow. A better approach is to provide to analytics users or creators a mechanism for specifying and implementing management actions that work well for their codes' characteristics. Our first design goal, therefore, is that *(1) management routines and policies should be customizable on a per container basis.*

To make management decisions at run-time, information is needed by management functions to determine when and what actions should be performed. The specific information collected and its organization depends on the management policies being enforced. This requires the continuous monitoring of workflow components, their behavior and running times, and of the physical resources they use, thus permitting management actions to be invoked in a timely manner. Our second design goal states that: *(2) management actions are guided by user-determined metrics driving per-container and cross-container (i.e., global) management policies.*

Ideally, analytics workflow pieces should be decoupled along the time and space dimensions so that a component's correct operation depends only on the availability of the necessary data (i.e., from disk or via the network). With well-defined input and output interfaces, analytics actions can be allowed to run independently as separate applications (i.e., components), and enter and leave the pipeline as needed. This makes it possible to run entirely different, dynamically swappable analytics codes without requiring them to be integrated into a single executable. Our next design goal, then, states that *(3) analytics codes should operate in a componentized fashion.*

A risk with management is that operations on one component can jeopardize the execution of other (e.g., dependent) components. For instance, consider the case of trading resources between two analytics components when recovering from some detected bottleneck. A failure can occur if there is an inconsistent view of the state of the resources in which a component tries to use a resource that has not yet been fully relinquished by another component. Our last design goal, therefore, states that: *(4) management operations must be reliable and be resilient to failure.*

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

2.2 Conceptual Model

2.2.1 Containers

A container, depicted in Fig. 2 allows analytics tasks to be embedded into a dynamically managed messaging and execution framework. The container's input and output interfaces are similar in concept to those used in modern Service Oriented Architectures (SOA). The container is comprised of a set of *active replicas* that perform analytics actions on incoming data and a *container manager* that oversees its execution.

Active Replicas. Unlike the replication techniques used in fault tolerant systems [13, 16], where replicas have identical internal states, active replicas in containers are key to obtaining scalable container operation: with traditional replication, each replica performs redundant computations on the same data items, whereas active replicas perform their computations on different epochs of data assigned to them. For the use-case discussed in Section 4.1, data is assigned to active replicas in a round-robin fashion. Using active replicas, a container manager can increase its degree of parallelism by spawning a new replica.

Container Manager. It oversees the execution of its active replicas, assigning resources to them and gathering and organizing the information needed to make runtime decisions about their number. Container managers also have custom implementations of a set of management primitives, described next, which allows them to respond to management requests from higher-level (global) managers in a manner customized to the codes they run.

A container manager provides metadata services for the active replicas it manages. As stated previously, users embed codes into an execution and messaging runtime, and workflows like pipelines are constructed by chaining together containers along their I/O paths, to allow direct container-container data exchanges without involving the storage system and/or intermediate consolidators. Managers support this by maintaining state about inter-container connectivity (i.e., endpoint contact information) and by issuing state-change notifications to neighboring container managers. Container managers can also store important in the case where active replicas may implement stateful functions.

2.2.2 Management Constructs

Hierarchical container management affords several benefits. First, such hierarchies can be scaled with ease [32]. Second, distinct per-container managers can offer customized management routines and separate their local, per-component management states from global state about entire I/O pipelines. Hierarchy also helps define management authority: a global manager is responsible for operations that re-organize entire workflows, whereas individual container managers (i) are responsible for operations affecting only their components and resources, and (ii) respond to management invocations from higher-level (global) managers.

The following core management primitives make it possible to construct higher-level management policies and operations:

- **Increase Container:** allocate more resources to a container with the goal of increasing a container's scalability.
- **Decrease Container:** deallocate resources to a container; useful when resources are scarce and some containers may be over-provisioned.
- **Offline Container:** remove all resources from a container and direct dataflow from upstream containers to disk; useful when it is no longer feasible to run a container online, i.e., there are insufficient resources to scale the container or there may be a network partition in a geographically distributed workflow.

While the per-container management actions listed above are invoked by a global manager, the concrete steps needed to execute these actions within a container can be customized on a per container basis. For example, when told to "increase" its degree of parallelism, a code that cannot operate on data epochs out of order could implement its "increase" operation by killing its existing active replica and spawning a new one with a greater MPI size, whereas another container could implement this by just spawning a new replica and adding it to the existing cohort.

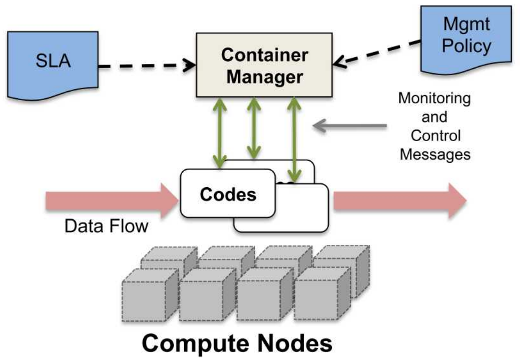


Figure 2: Container abstraction.

3. IMPLEMENTATION

3.1 Container

3.1.1 Active Replicas

The implementation of containers leverages the widely used ADIOS read and write interfaces [26]. Using these interfaces, analytics codes can specify their data requirements and establish communication via a virtual file name serving as a named communication channel. To accommodate active management, the Flexpath [11] ADIOS transport, which allows for online analytics routines to exchange output data, has been extended to accept and process management messages and state-change notifications from the replicas' designated container manager. We also modify the ADIOS interface to expose to analytics applications a communicator they can use to interact directly with the container manager, if needed.

There is additional queue management for Flexpath publishers (ADIOS writers): they maintain a queue for each neighboring Flexpath reader replica (in a downstream container) to hold epochs of data. Writers then assign data to these queues in a fashion determined by the reader container. The current implementation supports round-robin assignment, including the case in which one replica consumes all of the work for an existing replica, explained in more detail in Section 4. Management actions can also lead to queue management operations, as when upon the arrival of a new replica in a downstream container, load balancing actions reduce the lengths of overly filled queues. Conversely, with a 'decrease' operation, we can re-assign existing work to other remaining replicas.

3.1.2 Managers

Managers are written to be run as stand-alone executables. Users can create custom managers and specify SLAs using a programmatic API, described in Section 3.2. When global managers detect conditions of interest, they invoke management commands on container managers, and then distribute any important state changes to subsequent container managers that need to be aware of such state changes. Container managers are responsible for carrying out custom implementations of management commands invoked on them by global managers, and for performing internal actions on the resources and replicas they manage.

3.2 Management Interface

The basic primitives listed in Section 2.2.2 are exposed as a C interface, and developers use this interface to create custom managers. To meet a SLA at a global manager, management codes can read monitoring information, and then carry out chained primitives to perform actions like resource trading. When invoked, a management primitive triggers a set of transactional protocols that indicate a participant's progress and distribute any state changes.

The sample policy shown below triggers resource re-assignment across containers when the output queue for *container1* reaches a queue length of 10.

```
if (container1->max_queue_length == 10) {
  decrease_container(container2, 1);
  while (container2->state != STEADY_STATE)
    sleep(1);
  increase_container(container1, 1);
}
```

An 'increase' command received by a container manager, to add to its working size, may launch additional replicas. In our current implementation on the Titan machine at Oak Ridge National Labs, launching of replicas is conducted as follows: the container manager constructs an *aprun* command as a text string and then writes this command to a file. A *PBS* script (a feature of the PBS job scheduler), scans each container manager's file for commands, and when one is present, it reads and executes it. This implementation is due to the constraint that only the root node of the job, which executes the PBS script, can launch applications on the compute nodes. While this illustration and the use case presented in Sec. 4.1 focus on output queue build up, management could also be triggered by other factors, such as memory consumption or CPU utilization.

3.3 Container Information Bus

Monitoring, control, and state change messages are delivered via the *Container Information Bus*, or CIB, implemented using the EVPath [15] messaging library.

Managers and replicas are connected via the CIB's overlay graph, where container-level managers serve two roles: (1) aggregator nodes used to gather and organize monitoring information, execution metadata, and runtime state information before delivery to the global manager; and (2) as entry points into a container in terms of management operations and the delivery of state change notifications (i.e., state that determines from who replicas read data) from neighboring containers.

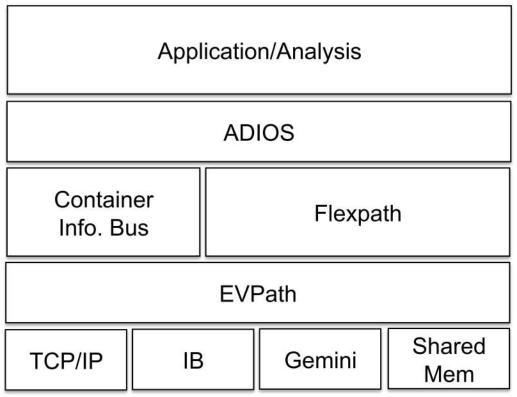


Figure 3: I/O Container software architecture.

Global managers serve as the root of the CIB and accept and organize messages from all container managers. In order to ensure the strong consistency of runtime state information, the current implementation passes all messages relating to state changes through the global manager. In the case of parallel replicas (i.e., MPI based analytics codes), we designate rank 0 as the recipient of messages from the container manager. Rank 0 then uses MPI to disperse the messages to the remaining ranks. We have done this to take advantage of MPI's optimizations and to also reduce the number of connections a container manager has to maintain.

3.4 Fault Detection and Recovery

The current implementation detects faults in two ways. The first uses application-level progress indicators delivered via periodic heartbeat messages from an application replica to its container-level manager. The second allows the manager to recieve a notification from the kernel when the socket between a manager and a replica has been disconnected. Method 1 does not rely on a specific messaging technology (sockets) and can work for a variety of underlying network interconnects, with the disadvantage that the manager must propagate this failure notification through the CIB to interested parties. Method 2 allows for any component interacting with it (managers, other replicas in the workflow) to receive the notification without having to wait for failure alerts to propogate through the CIB. Both methods are chosen for our current investigation, because they are familiar to end-users and have well-understood characteristics. Future work will explore more robust fault detection [29, 9] and diagnostic [32] mechanisms.

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

4. EXPERIMENTAL EVALUATION

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

The LAMMPS molecular dynamics simulation and the SmartPointer analysis toolkit serve as our application drivers for Titan, and we construct two policies to demonstrate the benefit of the I/O Container approach and to assess the overheads of its active management capabilities. We run the GTS machine on Maquis and execute the FFT code on a machine at a remote location, thereby allowing us to test the system's behavior when the workflow is geographically distributed. We note that we cannot conduct such experiments on Titan, as its security policies and firewall settings prevent us from doing so.

4.1 Using I/O Containers

4.1.1 *LAMMPS and SmartPointer*

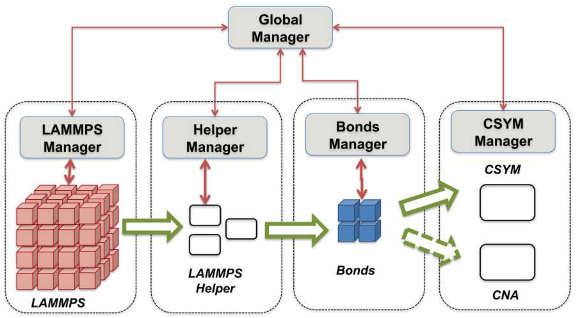


Figure 4: I/O Pipeline for LAMMPS with I/O Containers

Figure 4 depicts the I/O pipeline constructed for the LAMMPS (Large Scale Atomic/Molecular Massively Parallel Simulator) [30] science application, using the SmartPointer analysis and visualization toolkit. LAMMPS is a molecular dynamics simulation used across a number of science domains. It is written with MPI and performs force and energy calculations on discrete atomic particles. After a number of user-defined epochs, it outputs the atomistic simulation data (positions, atom types, etc.), with the size of this data ranging fom megabytes to terabytes depending on the science being conducted.

SmartPointer is a representative analytics pipeline interpreting LAMMPS output data to detect and then scientifically explore plastic deformation and crack genesis. In such scenarios, a force is applied to the material being simulated until it first starts to break. The SmartPointer set of analyses are configured to detect and categorize the geometry of the region around that initial break. SmartPointer implements functions to determine where and when plastic deformation occurs and to generate relevant information as the material is cracked. We summarize the SmartPointer codes in the list below, with additional detail found in [11, 12, 36]:

- *Lammps Helper*: parallel MPI code that serves as an aggregator and filter of the raw LAMMPS data.
- *Bonds*: parallel MPI code that performs an all-nearest neighbor calculation ($O(n^2)$) to label which atoms are bonded for each output epoch.
- *Csym*: a serial central symmetry analysis code that detects plastic deformation.
- *CNA*: a serial common neighbor analysis code that executes whenever CSYM determines that a deformation in the material has occurred. CNA is an extremely compute-intensive component ($O(n^3)$), and as such it should only execute when a crack has been detected in the material being modeled.

4.1.2 *GTS and FFT Analysis Code*

As an alternative application example, to demonstrate the more general utility of I/O Containers, we also evaluate our framework with GTS [34], a plasma fusion simulation with an implementation that exploits coarse grained process level parallelism using MPI, and more fine-grained thread-level parallelism using OpenMP. This "particle in cell" code has different output frequencies for both particles and mesh-level statistics. In order to examine the dynamics involved, in particular dangerous transient effects that might damage a real reactor vessel, it is useful to dynamically evaluate and characterize particular trends on the inner and outer edges of the plasma. Unlike the LAMMPS case, these transients are not as algorithmically identifiable, so secondary analysis methods are used to infer their existence, and then, much more detailed inspection involving direct interaction with the physicists is used to further the investigation. The GTS analytics pipeline is an FFT code that ingests the phi and Z-ion output arrays from the simulation.

4.2 Management Policies

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

Quality of Service (Global): the Bonds and CNA codes are slow components compared to the LAMMPS simulation, with CNA being the most expensive. Bonds executes on every output epoch, whereas CNA executes only when CSYM reports a crack. Depending on the output frequency of LAMMPS, or how soon a crack is detected, these codes can become a bottlenecks in the pipeline. We create a policy that monitors queue lengths such that if the global manager detects a growing queue length on some output container, and if a the queue size reaches a threshold, we perform an increase operation that results in spawning additional replicas for the slow component, which in this workflow, is either Bonds or CNA. This represents a global policy seeking to balance pipeline components to ensure healthy end-to-end throughput. It also allows for the workflow to run without needing to carefully provision both Bonds and CNA codes; the system can handle the provisioning when needed.

Data-centric (Local): requires application introspection into the data, based on the CSYM and CNA components. In contrast to the first policy, the metric of interest is reported by the analysis functions (when CSYM detects a crack), and the management actions (kill CSYM and run CNA) are triggered by the container-level manager. The goal of this policy is to ensure quality of data via correct execution of workflow analysis functions.

For the GTS & FFT workflow, we connect the analysis, running on an end user's machine, with the simulation codes over a wide area network. We evaluate system behavior in terms of container output latency and memory consumption, when faced with an unexpected component departure, e.g., when an end user terminates analysis. We evaluation three recovery options, all involving failure detection on the remote machine and spawning a recovery replica on the cluster with the simulation. The first option allows for data loss, while the second avoids it. With these two cases, the recovery replica is launched in response to a notification of a failure. The third policy takes advantage of over-provisioning, where the container spawns an additional FFT replica on the compute cluster, which remains idle until its container manager detects the failure.

We evaluate these three recovery options to demonstrate the flexibility of I/O containers and their management. If components need a guarantee on data, they can pay the costs for it, but less critical codes can avoid these extra costs by tolerating missing output epochs.

4.3 Quality of Data Policy and Microbenchmarks

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

Management costs are governed both by the inherent properties of the management methods chosen and their underlying protocols, and by the scales of interacting containers. The latter is due in part to the 'direct connect' nature of the Flexpath transport used in the implementation of I/O containers: Flexpath obtains high cross-container throughput by directly connecting the parallel entities of a previous container to the parallel entities of a subsequent one. This also means, however, that the cost of distributing certain state changes (e.g., container increase) is affected by the size of the neighboring containers, as each of their parallel entities must be notified about this state change.

Figure 5(a) shows the modest protocol overheads for an *increase* operation on the Bonds container. The bar titled "Helper Container" represents the time it takes for the Helper container to distribute the Bonds state change. This includes the time it takes for the container manager to send the state change to each replica (rank 0), and the time it takes for rank 0 to broadcast this change to the other ranks. The bar titled "Global to Local Managers" is the total time spent for all messages between the global manager and the container managers to trigger the management action, and to distribute the state changes. As expected, use of a management hierarchy allows for good scalability, demonstrated by the fact that for each point on the *x*-axis, we are increasing the number of Lammps Helper processes by a factor of 4, but only see a growth of 2*x* in terms of protocol cost. Since these management actions do not affect the number of managers, the communication between global and container-level is not affected by scale.

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

4.4 Throughput Measurements: QoS Policy

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

	LAMMPS	Helper	Bonds	CSYM
Fig 7(a)	8192	64	256 to 768	1
Fig 7(b)	4096	32	128 to 384	1
Fig 7(c)	2048	16	64 to 192	1

Table 1: Core Counts for Throughput Experiments

Fig. 6 shows the baseline, unmanaged workflow execution, for a LAMMPS simulation running on 8192 cores and a pipeline comprised of 64 Lammps Helper cores, 256 Bonds cores, and 1 CSYM core. The graph shows that as the output queue for Lammps Helper fills up, LAMMPS' throughput drops significantly. This is because it has to block on its output actions that must wait on queue space to free up. LAMMPS' throughput converges to that of Bonds, the slow component, effectively dropping end-to-end throughput to a third of the ideal target.

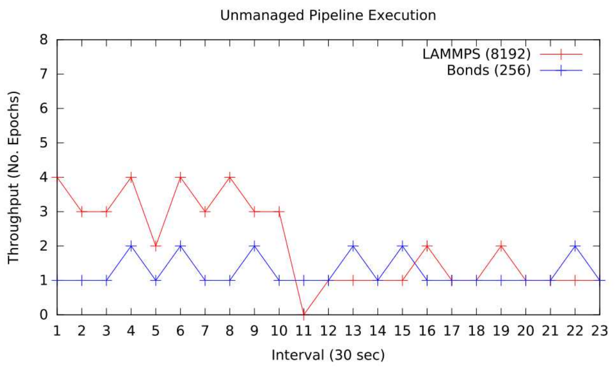
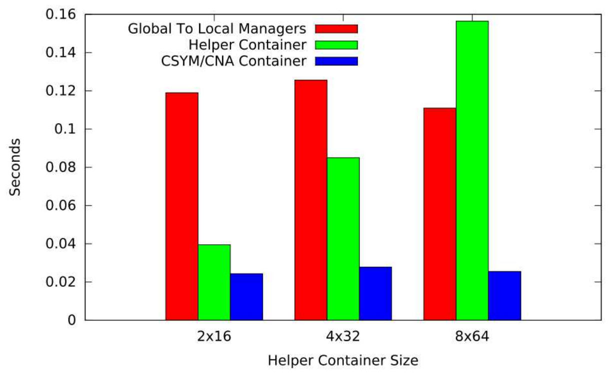


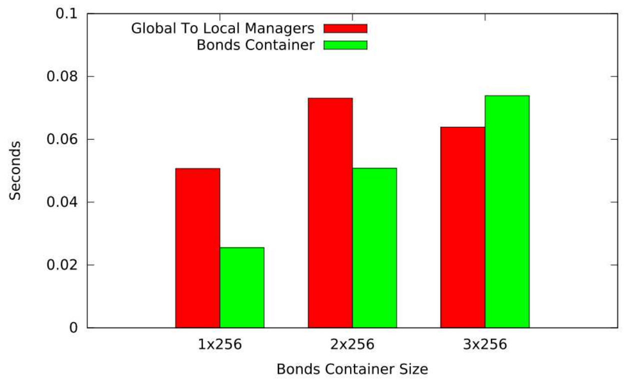
Figure 6: Throughput degradation for unmanaged pipeline.

Fig. 7 depicts the throughput improvements for a set of QoS-managed runs that demonstrate the container runtime's ability to provide elasticity at scale. Experiments are run at three scales, with the process counts displayed in Table 1. For each experiment, the slow container is increased by a replica with the number of processes equal to the size of the initial replica. For these runs, the crack in the material did not materialize until the end of the run, so that the main component needing an increase was the Bonds code. Fig. 7(a) shows the throughput improvements for running with 8192 Lammps cores. The vertical lines represent when Bonds is increased. For this run, we see that after the first increase (two Bonds replicas total), we see an improvement in Bonds throughput. However, an additional increase is needed for Bonds to match the throughput of the LAMMPS simulation. After this second increase (3 Bonds replicas, 768 cores total), we see that Bonds can achieve a higher throughput than the LAMMPS application, as it now has sufficient resources to start to drain the data that has built up in the queue.

Figure 7(b) shows a similar result, where after three increases, Bonds maintains a slightly higher throughput than the LAMMPS simulation. However, speedup is insufficient to fully drain the queue



(a) 'Increase' container command protocol overhead.



(b) Data Centric command protocol overhead assessment.

Figure 5: Protocol overhead measurements

in Lammps Helper, so the Bonds code executes somewhat longer. We see a similar phenomenon in Fig. 7(c), where the global manager does not increase the Bonds container by an additional replica because the stated policy is to trigger an increase only when two conditions are met: (1) a maximum queue length of 10 in one of the Helper output queues, and (2) a growing maximum queue length for 3 consecutive measurements. For the latter two runs, condition (2) did not trigger. This example illustrates the utility of explicit policy specification. An alternative policy omitting the second condition would have triggered the additional Bonds increase. An energy-conscious policy might prefer a slight extension in execution time over the additional energy consumed by using additional nodes.

Fig. 8 displays the changing queue length, the metric on which we base throughput management, for an experiment with the same setup as in Fig. 7(a). This represents the maximum queue length in the Lammps Helper container's output queue for the Bonds container. Here, the *x* axis represents the output epoch, and the *y* axis represents the max queue count when that output epoch is inserted into a queue. As is evident, the stated management policy is having the desired effect on its metric of interest.

4.5 Fault Recovery Policy

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

Fig. 9 displays the changes in container latency for three different fault-recovery mechanisms. The *x*-axis represents the epoch number for a container, and the *y*-axis represents the length of time between a step and the previous step. The first time step for each has a high latency, since we use the application start time as the base.

The first graph, Fig. 9(a), shows the container latency when recovering from a fault, but allowing for data loss, which is represented by the discontinuity for the FFT line. This has the lowest latency across all three because the previous (in other words, the older) time steps are simply dropped. Allowing for dropped epochs of data becomes more even more beneficial with configurations where it is infeasible, in terms of memory requirements, to buffer multiple timesteps of data.

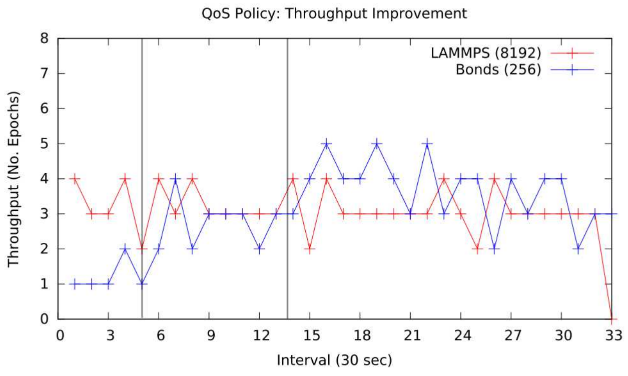
The second and third graphs show the changes in container latency when avoiding data loss. As expected, we see a higher container latency than when allowing for data loss as the older timesteps stay in the queue. The third graph has a lower container latency during the failure and recovery phases, because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

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

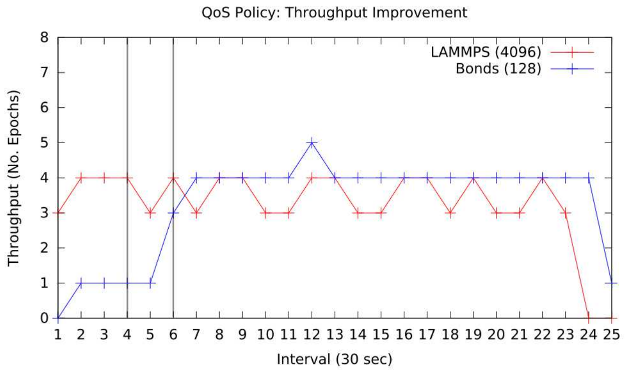
4.6 Discussion

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

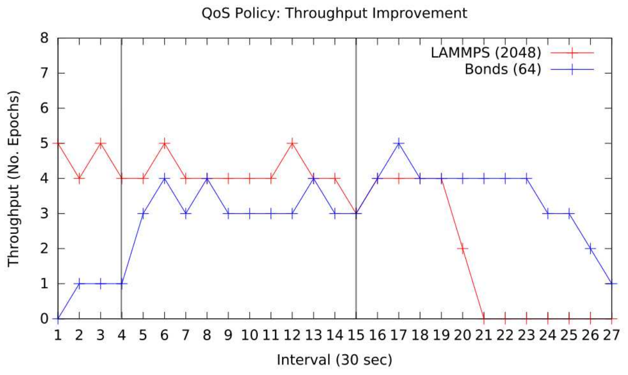
Container-based management scales through use of a hierarchical approach, permitting for (i) per-run customization of management policies and SLAs, (ii) specification and enforcement of such policies at different granularities in the workflow, and (iii) scalable



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



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



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

Figure 7: QoS Policy: throughput improvements.

implementations of management protocols, including those offering high reliability in management.

The performance results shown above demonstrate the superiority of managed vs. unmanaged I/O, guided by simple policies realized with low cost management structures. While able to scale to the high end machines currently available to our research, the current management policies implemented for containers assume each container running on its own dedicated resources, separate from those used by the application. Management actions that involve scheduling or resource sharing [41] remain part of our future work.

5. RELATED WORK

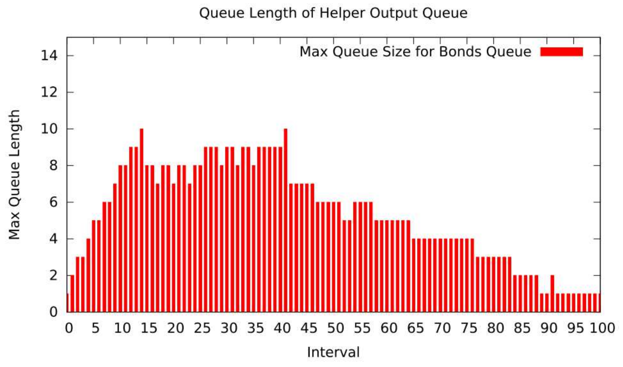


Figure 8: Change in max queue length for Helper Container.

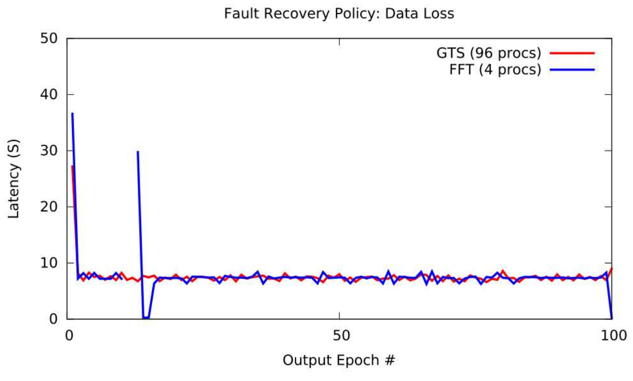
While currently realized for ADIOS-based I/O pipelines, containers are equally useful to other 'data staging' solutions, as long as they describe and use well-defined component interfaces. However, without using actual virtualization solutions like Palacios [24], additional programming and integration efforts will be required to add the concept to tightly integrated analytics codes in which individual actions are not separately defined and/or use well-defined component APIs, such as the analysis pipelines constructed with compiler-based systems like IBM's System S.

Hierarchical management is commonly used in enterprise systems [32]. I/O containers differ in that they explicitly address the parallel and high performance nature of components in the I/O and analytics pipelines run for high end simulations. Similarly, while one may view a containers as a limited form of hypervisor controlling the execution of its components, containers are not concerned with running multiple such entities and isolating them in terms of performance or for security purposes. Therefore, their functionality is perhaps, more akin to that of 'resource islands' explored for high end multicore processors [31], but they differ from such prior work in the explicit specification of container management policies and actions, exposed to end users and in enabling custom and application-specific methods for managing the analytics and visualization components present in I/O pipelines.

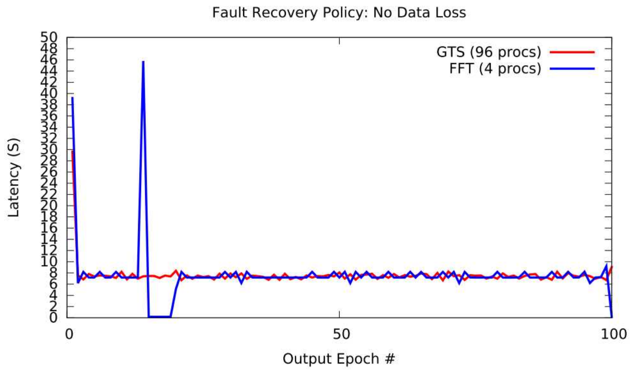
I/O containers predate but are similar in notion to the 'containers' now offered in Linux-based systems, developed for datacenter application [19]. They differ in their focus on managing parallel applications, at scale, rather than dealing with the fine grain, per-machine resource sharing targeted by Linux containers. Mesos' original intent was to permit fine grain resource sharing across multiple applications running in datacenter systems with only recent work (unpublished) exploring resource trading methods that may also be suitable for the HPC domain.

Other HPC-centric work on managing the analytics and visualization workflows on high end machines [20] provides adaptation policies at different layers of the stack (cross-layer adaptation), targeting an adaptive mesh refinement (AMR) code. It focuses on specific policies at different layers, to ensure minimal time to solution, whereas our work investigates the mechanics and abstractions of management that would be suitable for analytics workflows; the policies discussed in [20] are examples of additional policies suitable for implementation with the Containers framework.

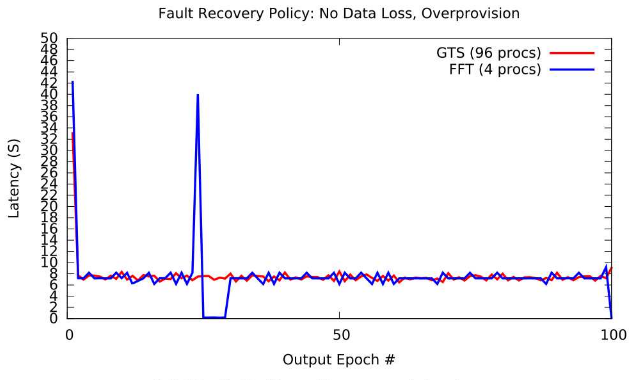
Our own earlier work on 'service augmentation' [35] demonstrates the utility of attaching Quality of Service (QoS) management actions to I/O pipelines and shows that container principles can be applied to other data staging or streaming infrastructures and systems, including [14] and [18], both of which use compo-



(a) Fault Policy: Data Loss



(b) Fault Policy: No Data Loss



(c) Fault Policy: Overprovisioning

Figure 9: QoS Policy: throughput improvements.

nentized approaches.

6. CONCLUSIONS AND FUTURE WORK

The I/O Containers framework presented in this paper permits users to embed their scientific data analytics tasks into a dynamically managed execution environment that (1) continually monitors analytics components for metrics of interest, (2) allows users to specify management policies and enforcement operations at different granularities of the workflow, (3) provides elasticity at scale for their analytics tasks, and (4) does so efficiently with low management overheads. The utility of I/O containers is demonstrated with three policies associated with I/O pipelines consisting of realistic science applications and analytics pipelines: (1) a global 'quality of service' policy permits an I/O pipeline to recover from a poor initial resource allocation; (2) a 'quality of data' policy operating at container level allows for new analytics tasks to be injected into the pipeline to respond to the richness of features discovered in the data; and (3) fault recovery policies handle an unexpected component departure in a geographically distributed workflow.

Our future work will address two different dimensions of container-based management. One is to gain broader insights into the resilience issues associated with online management, exploring the robust failure mechanisms developed in previous work in a science workflow setting. Another task for future work is to better understand management in environments where analytics operate 'in situ' with simulations, leading to management actions that involve fine grain resource sharing and scheduling [41] and giving rise to concerns with performance isolation [24, 5].

7. ACKNOWLEDGEMENTS

This research was supported by the Department of Energy Office of Advanced Scientific Computing Research. It also used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000. SAND2014-19011 C

8. REFERENCES

[1] Hadoop: http://hadoop.apache.org.

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

[3] *2005 IEEE International Conference on Cluster Computing (CLUSTER 2005), September 26 - 30, 2005, Boston, Massachusetts, USA*. IEEE, 2005.

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

[5] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. L. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In M. L. Scott and L. L. Peterson, editors, *SOSP*, pages 164–177. ACM, 2003.

[6] J. Borgdorff, M. Mamonski, B. Bosak, K. Kurowski, M. B. Belgacem, B. Chopard, D. Groen, P. V. Coveney, and A. G. Hoekstra. Distributed multiscale computing with MUSCLE 2, the multiscale coupling library and environment. *CoRR*, abs/1311.5740, 2013.

[7] D. Boyuka, S. Lakshminarasimhan, X. Zou, Z. Gong, J. Jenkins, E. R. Schendel, N. Podhorszki, Q. Liu, S. Klasky, and N. F. Samatova. Transparent in situ data transformations in ADIOS. In *CCGrid '14*. IEEE.

[8] R. Brightwell, T. Hudson, K. T. Pedretti, R. Riesen, and K. D. Underwood. Implementation and performance of portals 3.3 on the cray XT3. In *2005 IEEE International Conference on Cluster Computing (CLUSTER 2005), September 26 - 30, 2005, Boston, Massachusetts, USA* [3], pages 1–10.

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

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

[11] J. Dayal, D. Bratcher, G. Eisenhauer, K. Schwan, M. Wolf, X. Zhang, H. Abbasi, S. Klasky, and N. Podhorszki. Flexpath: Type-based publish/subscribe system for large-scale science analytics. In *CCGrid '14*, pages 246–255.

[12] J. Dayal, J. Cao, G. Eisenhauer, K. Schwan, M. Wolf, F. Zheng, H. Abbasi, S. Klasky, N. Podhorszki, and J. F. Lofstead. I/O containers: Managing the data analytics and visualization pipelines of high end codes. In *HPDIC '13*, pages 2015–2024. IEEE, 2013.

[13] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[14] C. Docan, M. Parashar, and S. Klasky. DataSpaces: an interaction and coordination framework for coupled simulation workflows. In *HPDC 2010*. ACM.

[15] G. Eisenhauer, M. Wolf, H. Abbasi, and K. Schwan. Event-based systems: opportunities and challenges at exascale. In *DEBS*, 2009.

[16] K. B. Ferreira, J. Stearley, J. H. L. III, R. Oldfield, K. T. Pedretti, R. Brightwell, R. Riesen, P. G. Bridges, and D. Arnold. Evaluating the viability of process replication reliability for exascale systems. In S. Lathrop, J. Costa, and W. Kramer, editors, *SC '11*. ACM.

[17] E. R. Hawkes, R. Sankaran, J. C. Sutherland, and J. H. Chen. Direct Numerical Simulation of Turbulent Combustion: Fundamental Insights Towards Predictive Models. *Journal of Physics: Conference Series*, 16(1):65, 2005.

[18] M. Hereld, M. E. Papka, and V. Vishwanath. Toward Simulation-Time Data Analysis and I/O Acceleration on Leadership-Class Systems. In *IEEE Symposium on Large-Scale Data Analysis and Visualization*, 2011.

[19] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: a platform for fine-grained resource sharing in the data center. In *NSDI '11*, pages 22–22, Berkeley, CA, USA. USENIX Association.

[20] T. Jin, F. Zhang, Q. Sun, H. Bui, M. Parashar, H. Yu, S. Klasky, N. Podhorszki, and H. Abbasi. Using cross-layer adaptations for dynamic data management in large scale coupled scientific workflows. In W. Gropp and S. Matsuoka, editors, *SC '13*. ACM.

[21] Y. Kanemasa, Q. Wang, J. Li, M. Matsubara, and C. Pu. Revisiting performance interference among consolidated n-tier applications: Sharing is better than isolation. In *2013 IEEE International Conference on Services Computing, Santa Clara, CA, USA, June 28 - July 3, 2013*, pages 136–143. IEEE, 2013.

[22] S. Klasky, S. Ethier, Z. Lin, K. Martins, D. McCune, and R. Samtaney. Grid-Based parallel data streaming implemented for the gyrokinetic toroidal code. In *SC '03*, page 24, Washington, DC, USA. IEEE Computer Society.

[23] P.-S. Koutsourelakis. Accurate Uncertainty Quantification Using Inaccurate Computational Models. *SIAM J. Scientific Computing*, 31(5):3274–3300, 2009.

[24] J. R. Lange, K. T. Pedretti, T. Hudson, P. A. Dinda, Z. Cui, L. Xia, P. G. Bridges, A. Gocke, S. Jaconette, M. Levenhagen, and R. Brightwell. Palacios and kitten: New high performance operating systems for scalable virtualized and native supercomputing. In *IPDPS*, pages 1–12. IEEE, 2010.

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

[26] J. Lofstead, F. Zheng, S. Klasky, and K. Schwan. In *Adaptable, metadata rich IO methods for portable high performance IO*, pages 1 –10, may 2009.

[27] D. Moise, G. Antoniu, and L. Bougé. Improving the hadoop map/reduce framework to support concurrent appends through the blobseer BLOB management system. In S. Hariri and K. Keahey, editors, *HPDC '10*. ACM.

[28] K. Moreland and D. Thompson. From cluster to wall with vtk. In A. H. J. Koning, R. Machiraju, and C. T. Silva, editors, *IEEE Symposium on Parallel and Large-Data Visualization and Graphics*, pages 25–32. IEEE, 2003.

[29] D. Peng, F. Dabek, and G. Inc. Large-scale incremental processing using distributed transactions and notifications. In *9th USENIX Symposium on Operating Systems Design and Implementation*, pages 4–6.

[30] S. Plimpton, R. Pollock, and M. Stevens. Particle-Mesh Ewald and rRESPA for Parallel Molecular Dynamics Simulations. In *PPSC*. SIAM, 1997.

[31] P. Tembey, A. Gavrilovska, and K. Schwan. intune: Coordinating multicore islands to achieve global policy objectives. In *TRIOS*. ACM, 2013.

[32] C. Wang, K. Schwan, V. Talwar, G. Eisenhauer, L. Hu, and M. Wolf. A flexible architecture integrating monitoring and analytics for managing large-scale data centers. In H. Schmeck, W. Rosenstiel, T. F. Abdelzaher, and J. L. Hellerstein, editors, *ICAC '11*, pages 141–150. ACM, 2011.

[33] G. Wang and T. S. E. Ng. The impact of virtualization on network performance of amazon EC2 data center. In *INFOCOM*, pages 1163–1171. IEEE, 2010.

[34] W. X. Wang, Z. Lin, W. M. Tang, W. W. Lee, S. Ethier, J. L. V. Lewandowski, G. Rewoldt, T. S. Hahm, and J. Manickam. Gyro-Kinetic simulation of global turbulent transport properties in tokamak experiments. *Physics of Plasmas*, 13(9):092505, 2006.

[35] M. Wolf, H. Abbasi, B. Collins, D. Spain, and K. Schwan. Service Augmentation for High End Interactive Data Services. In *CLUSTER* [3], pages 1–11.

[36] M. Wolf, Z. Cai, W. Huang, and K. Schwan. SmartPointers: Personalized Scientific Data Portals in Your Hand. In *SC*, pages 1–16, 2002.

[37] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI 12*, pages 15–28, San Jose, CA. USENIX.

[38] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In E. M. Nahum and D. Xu, editors, *2nd USENIX Workshop on Hot Topics in Cloud Computing, HotCloud'10, Boston, MA, USA, June 22, 2010*. USENIX Association, 2010.

[39] F. Zhang, C. Docan, H. Bui, M. Parashar, and S. Klasky. Xpressspace: a programming framework for coupling partitioned global address space simulation codes. *CCPE*, 26(3):644–661, 2014.

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

Machines.

[41] F. Zheng, H. Yu, C. Hantas, M. Wolf, G. Eisenhauer, K. Schwan, H. Abbasi, and S. Klasky. Goldrush: resource efficient in situ scientific data analytics using fine-grained interference aware execution. In W. Gropp and S. Matsuoka, editors, *SC '13*. ACM, 2013.