This paper describes objective technical results and analysis. Any subjective views or opinions that might be expressed in the paper do not necessarily represent the views of the U.S. Department of Energy or the United States Government.

SAND2014-19011C

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

Jai Dayal*, Jay Lofstead†, Greg Eisenhauer*, Karsten Schwan*, Matthew Wolf *‡, Hasan Abbasi‡, Scott Klasky‡
* *Georgia Institute of Technology*
† *Sandia National Laboratories*
‡ *Oak Ridge National Laboratory*

***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 science phenomena being simulated, a key challenge for online analytics is to respond to dynamics in workflow behavior 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 to respond to runtime dynamics at different granularities of the analytics workflow.

***Keywords*-Data Staging, Data Analytics, in-Situ, Visualization, Scalable I/O, Runtime Management, resource sharing**

I. Introduction

On current generation petascale platforms, scientific applications like the GTC [?] and S3D [?] simulations are already generating terabytes of data every few minutes. The desire to scale their 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 their large data volumes, without overwhelming the parallel file systems attached to these machines. These include running analytics along side simulations – "in-situ" [?], [?] – and in I/O staging areas – "in-transit" [?], [?], [?] – 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 [?], (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, it is projected that high-end codes will no longer be structured as a single large synchronous application, but rather, as a set of componentized codes running concurrently with the simulation that ingest and operate on its 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 substantial dynamics in their execution behavior, in part due to their data-dependent functionality, an example being an analytics code whose runtime is determined by the number of features found in the output data it analyzes. I/O pipelines, therefore, can experience dynamic changes in their resource consumption and requirements, making their initial resource allocations inappropriate and/or requiring adjustments in how analytics operate, e.g., through reductions in their precision or similar measures. 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 [?], [?] systems and for the multi-tier services run by web companies [?], [?].

The failure to react to online changes in the behavior of I/O pipelines can be severe, as 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, preventing 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 [?], [?]. 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

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

in Fig. ??, 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 [?], 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 for 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 explored elsewhere 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 [?]. A simple example is a guarantee that a resource removed from one container is successfully given to another.

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 division of management is one in which a global manager is responsible for maintaining an entire workflow's SLA, e.g., 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, ParaView [?], while another may run analytics using VTK [?]. A dynamic requirement for additional resources to run VTK'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.

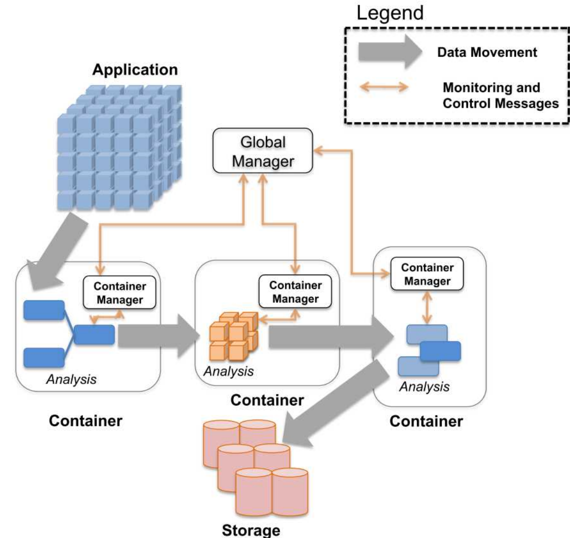


Figure 1: High-level view of I/O Containers framework.

I/O containers with their well-defined component interfaces and a programmatic management API expose to end users a set of basic management primitives for specifying their SLAs, by indicating the appropriate actions to take when certain conditions are detected. Condition detection operates at runtime, for individual containers and delivered to the management hierarchy via continuous online monitoring profiling individual containers for 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 [?], [?], able to run on cluster machines and on high end supercomputers like ORNL's Titan. Using the LAMMPS molecular dynamics simulation [?], along with the SmartPointer visualization and analysis toolkit [?], we evaluate I/O containers with SLAs that include: (1) a global performance-driven SLA that implements "elastic containers" to recover from detected bottlenecks in the I/O pipeline; and (2) a container-level data-centric policy that executes specific SmartPointer analysis routines only when certain features are detected in the output data being processed. Experimental evaluations show that the use of active container-based management can:

1) achieve elasticity at scale for representative science analytics;
2) create and enforce SLAs at multiple levels of an I/O pipeline; and
3) operate at large scales with low overheads.

An example demonstrating the utility of container-managed I/O shown in our evaluation uses a performance-driven SLA

to recover from a poor initial resource allocation and avoid degregaded throughput, resulting in a near 300% increase in end-to-end pipeline throughput compared to the unmanaged case.

I/O containers constitute new functionality in the domain of scientific data management, where 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 our own 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 [?]. Similarly, our recent work on 'in-situ' analytics for supercomputer simulations [?], schedules and manages only the analytics actions taking place on individual compute nodes, not being concerned about the end-to-end properties of the I/O pipelines originating at such nodes. Our future work, therefore, will start with useful policies for controlling combined in-situ and distributed I/O pipelines, but then also consider end-to-end goals like pipeline failure resilience and energy efficiency, supported by efficient node-level and/or component-level control loops.

II. 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 an executable that carries out analytics tasks on the data it ingests. More complex structures, like entire I/O pipelines, are supported by chains of containers supervised by a higher level manager interacting with per-container managers. This multi-level management scheme can maintain both container-level and global (i.e., across all containers) properties. 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 made available by the container and controlled by potentially container-specific management and scheduling. Fig. **??** depicts a conceptual model of I/O containers.

A. 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 [?], [?], [?] in terms of their data characteristics, execution models, and degrees of parallelism.

- **Functional Dependencies.** Analytics codes expect to ingest data that matches specific formats and layouts, and analytics functions may also need to transform the 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.** Science codes are often heterogeneous in terms of their parallelism, execution models, fault tolerance, and scaling characteristics, resulting in substantial variations across different I/O containers and the components they manage.
- **Stringent Resource Constraints.** Resources are not free, with the bulk of the resources typically assigned to the simulations being run, whereas analysis codes are given 'spare' resources, i.e., spare CPU cycles on simulation nodes [?], [?], reserved staging nodes [?], [?], 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 we have outlined in the previous section, we formulate the following design goals for I/O container-based workflow management. 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.*

In order to make management decisions at run-time, information is needed by management functions to determine when and what actions should be performed, with the specific information collected and its organization depending 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, pieces of the analytics workflow 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.

B. Conceptual Model

1) Containers: A container, depicted in Fig. **??** 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 [**?**], [**?**], 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 our use-case discussed in section **??**, 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, by 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 also provides metadata services for the active replicas it manages. As stated previously, users embed codes into an execution and messaging runtime, and pipelines or workflows are constructed by chaining together containers along their I/O paths, to allow direct container-container data exchanges without involving the storage system. 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) Management Constructs: Hierarchical container management affords several benefits. First, such hierarchies can be scaled with ease [**?**]. 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 permit construction of 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.

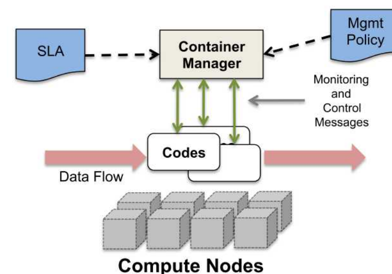


Figure 2: Container abstraction.

III. Implementation

A. Container

1) Active Replicas: The implementation of containers leverages the widely used ADIOS read and write interfaces [**?**]. Using these interface, 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 [**?**] 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.

We added queue management for Flexpath publishers (ADIOS writers) 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 **??**. 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.

2) Managers: Managers are written to be run as standalone executables. Users can create custom managers and specify SLAs using a programmatic API, described in Section **??**. 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.

B. Management Interface

We have exposed the basic primitives listed in Section **??** as a C interface, and developers use this interface to create custom managers. To specify an SLA at a global manager, developers can read monitoring information, and then chain these commands together to perform actions such as resource trading. When invoked, the management primitives trigger a set of transactional protocols that indicate a participant's progress and distribute any state changes.

An 'increase' command received by a container manager, to add to its working size, may launch additional replicas. In our current implementation, based 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 it and then 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 **??** focus on output queue build up, management could also be triggered by other factors, such as memory consumption or CPU utilization.

C. Container Information Bus

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

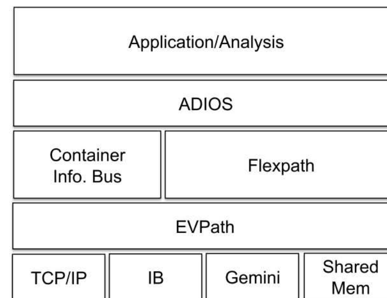


Figure 3: I/O Container software architecture.

Managers and replicas are connected via the Container Information Bus'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.

Global managers serve as the root of the Container Information Bus 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.

IV. Experimental Evaluation

All experimental evaluations are conducted on the Titan supercomputer hosted at Oak Ridge National Labs. 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 LAMMPS molecular dynamics simulation and the SmartPointer analysis toolkit serve as our application drivers, and we construct two policies to demonstrate the benefit of the I/O Containers approach and to assess the overheads of performing management.

A. Use Case

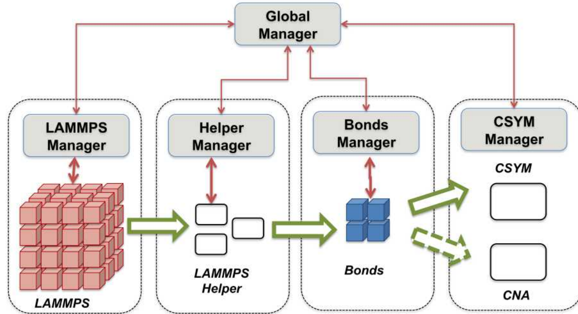


Figure 4: I/O Pipeline for LAMMPS with I/O Containers

Figure **??** depicts the I/O pipeline constructed for the LAMMPS (Large Scale Atomic/Molecular Massively Parallel Simulator) [**?**] 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 [**?**], [**?**], [**?**]:

- *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.

B. Management Policies

Two sample management policies serve the needs of the LAMMPS I/O pipeline used in our evaluation:

1) *Quality of Service (Global)*: the Bonds code is a slow component compared to the LAMMPS simulation, and since it executes on every output epoch, it can become a bottleneck in the pipeline. We create a policy that monitors queue lengths such that if the global manager detects a growing queue length for one of Lammps Helper's output queues, and if a the queue size reaches a threshold, we perform an increase operation that results in spawning additional Bonds replicas. This represents a global policy seeking to balance pipeline components to ensure healthy end-to-end throughput.
2) *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 correct execution of the workflow analysis functions.

C. 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.

Fig. **??** 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, as it has to then block on its output actions that must wait on queue space to free up. LAMMPS' throughput converges to that of Bonds, the slow component, effectively dropping end-to-end throughput to a third of the ideal target.

Fig. **??** depicts the throughput improvements for a set of QoS-managed runs that demonstrate the container runtime's ability to provide elasticity at scale. We have run experiments at three scales, with the process counts displayed in Table **??**. For each experiment, the Bonds container is increased by a replica with the number of processes equal to the size of the initial replica. Fig. **??** shows the throughput improvements for running with 8192 Lammps

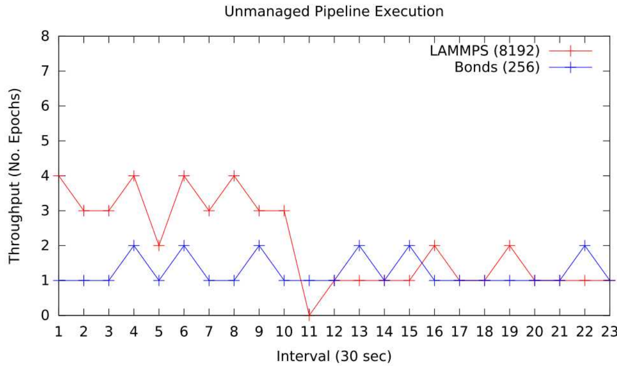


Figure 5: Throughput degradation for unmanaged pipeline.

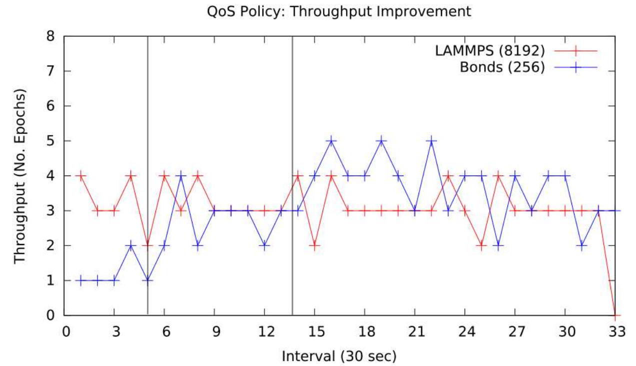
	LAMMPS	Helper	Bonds	CSYM
Fig **??**	8192	64	256 to 768	1
Fig **??**	4096	32	128 to 384	1
Fig **??**	2048	16	64 to 192	1

Table I: Core Counts for Throughput Experiments

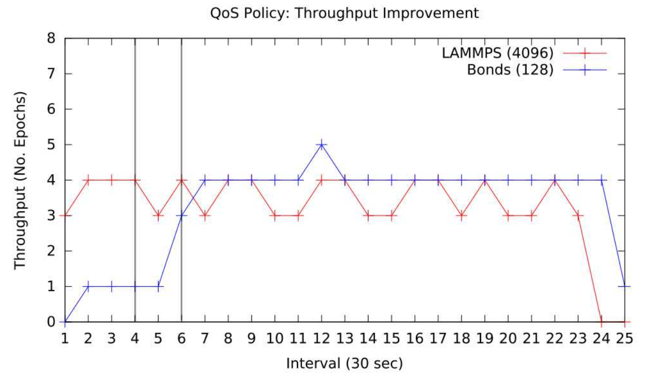
cores. The vertical lines represent when Bonds is increased. For this run, we see that after the first increase (two Bonds replicas total), we see an improvement in Bonds throughput. However, an additional increase is needed for Bonds to match the throughput of the LAMMPS simulation. After this second 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 **??** shows a similar result, where after three increases, Bonds maintains a slightly higher throughput than the LAMMPS simulation. Here, however, speedup is insufficient to fully drain the queue in Lammps Helper, so the Bonds code executes somewhat longer. We see a similar phenomenon in Figure **??**, where the reason the global manager does not increase the Bonds container by an additional replica is because the stated policy is to trigger an increase 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) didn't 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.

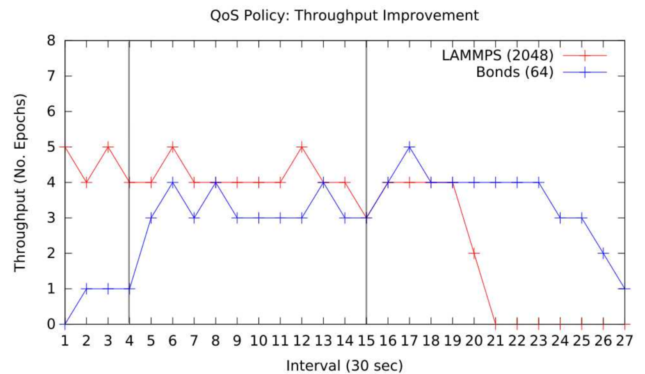
Fig. **??** displays the changing queue length, the metric on which we are basing our throughput management, for an experiment with the same setup as Fig. **??**. This represents the maximum queue length in the Lammps Helper container's output queue for the Bonds container. Here, the x axis represents 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.



(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

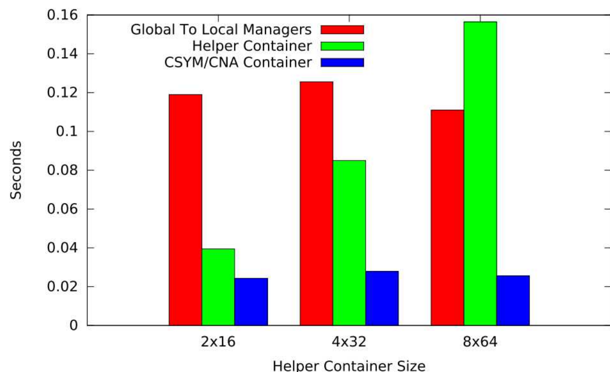


(c) 2048 LAMMPS cores with 1 to 4 Bonds replicas of size 64

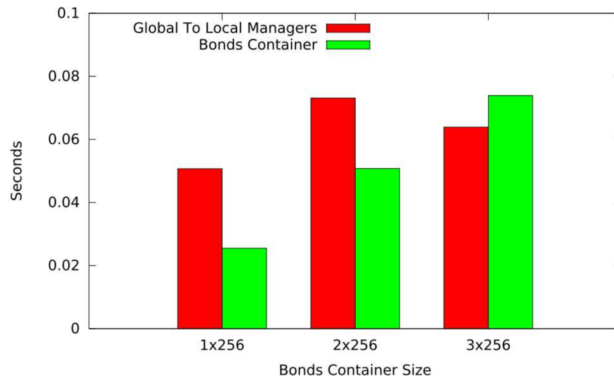
Figure 6: QoS Policy: throughput improvements.

D. 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 two 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.



(a) 'Increase' container command protocol overhead.



(b) Data Centric command protocol overhead assessment.

Figure 8: Protocol overhead measurements

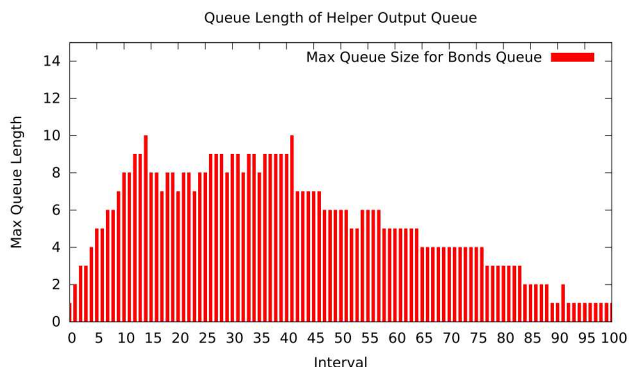


Figure 7: Change in max queue length for Helper Container.

Management costs are governed both by the inherent properties of the management methods chosen and by container 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 **??** shows the modest protocol overheads for an *increase_container* 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 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 $2x$ 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. **??** shows the cost of the protocol used to enforce the data-centric management policy. This represents a control loop that is 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.

E. Discussion

Container-managed I/O pipelines provide elasticity at scale 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 the individual analysis components.

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 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 run on the high end machines currently available to our research, the current management policies

implemented for containers assume each container to run on its own dedicated resources, separate from those used by the application. Management actions that involve scheduling or resource sharing [?] remain part of our future work.

V. Related Work

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, 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 analytics pipelines constructed with compiler-based systems like IBM's System S.

Hierarchical management is commonly used in enterprise systems [?]. 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 component, containers are not concerned with running multiple such entities and isolate them in terms of performance or for security purposes, but their functionality is perhaps, more akin to that of 'resource islands' explored for high end multicore processors [?]. Also different from such prior work is the explicit specification of containers' management policies and actions, exposed to end users and 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 [?]. 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.

Other HPC-centric work on managing the analytics and visualization workflows on high end machines [?] provides adaptation policies at different layers of the stack (cross-layer adaptation), targeting an adaptive mesh refinement (AMR) code. Such work complements our research and may lead to additional useful management policies embedded in containers.

Our own earlier work on 'service augmentation' [?] 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 [?] and [?] both of which use componentized approaches.

VI. Conclusions and Future Work

The I/O Containers framework presented in this paper allows for 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 two policies associated with a realistic I/O pipeline run with a representative high end simulation, the LAMMPS molecular modeling code: (1) a global 'quality of service' policy permit an I/O pipeline to recover from a poor initial resource allocation, effectively improving its end-to-end throughput by nearly 300%, and (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.

Our future work will address two different dimensions of container-based management. One is to gain insights into the resilience issues associated with online management, where failures can happen at any stage during the execution of a management operation. Our initial work in that space has explored transactional constructs [?]. Another 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[?] and giving rise to concerns with performance isolation[?], [?].

VII. Acknowledgements

This research was supported by the Department of Energy Office of Advanced Scientific Computing Research. This research 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.