2015 IEEE 11th International Conference on eScience

SODA: Science-driven Orchestration of Data Analytics

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*—As scientific simulation applications evolve on the path towards exascale, a new model of scientific inquiry is required where concurrently with the running simulation, online analytics operate on the data it produces. By avoiding offline data storage except when absoluately necessary, it enables speeding up the scientific discovery process by providing rapid insights into the simulated science phenomena and affording more frequent, detailed data analytics than is possible with the traditional purely offline approach of using disk for intermediate data storage. However, a challenge for online analytics is to respond to behavior dynamics caused by changing simulation outputs and by unforeseen events on the underlying hardware/software platforms.**

This paper presents SODA, a set of run-time abstractions for online orchestration of data analytics, realized by embedding analytics tasks into workstations that monitor component behavior and enable responses to run-time changes in their resource demands and in the platform's resource availability.

For high end simulations running on a leadership class machine, experimental evaluations show SODA can invoke efficient orchestration operations responding to a diverse set of run-time dynamics at different granularities to meet end-user and analysis specific requirements.

***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 [1] fusion and S3D [2] combustion simulations are already generating terabytes of data every few minutes. Scaling the analytics and visualization codes for such data volumes at these output frequencies has required researchers to devise new online methods to avoid overwhelming the parallel file systems attached to these machines. While new memory hierarchy layers such as non-volatile memory in the compute area help address the I/O bandwidth mismatch, it only shifts the problem to a higher performing, but much more limited resource rather than addressing it. Instead, different methods must be enabled. These include running analytics concurrently along side simulations – "in-situ" [3], [4] – and in I/O staging areas – "in-transit" [5], [6] – 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 (1) continuously ascertaining simulation validity permitting termination or correction without unduly wasting machine resources [7], (2) gaining rapid insights into the scientific processes being simulated (online visualization), or even (3) enabling methods for application steering. Projections suggest accommodating all of these features will lead high-end codes to be structured as a set of concurrently running components continuously processing the simulation data rather than as a single, large, synchronous application integrating through the file system. 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, analytics codes present considerable variations in I/O pipelines. They differ in features such as their maturity, degrees of parallelism, execution models, data characteristics, and resilience capabilities. They can also exhibit dynamic execution behavior driven by the data itself. For example, an analytics code's runtime may be determined by the number of features found in the data it analyzes. I/O pipelines, therefore, may dynamically change resource consumption and requirements making their current resource allocations inappropriate and/or require adjusting how analytics operate to avoid over-provisioning for the maximum resource requirement case. Failure to react to changes in I/O pipeline behavior can lead to severe consequences. Unduly slow analytics pipelines can cause data loss or stall high end simulations by causing them to block on their output actions [5], [8], [9], [10].

This paper describes the *SODA* approach to managing dynamic I/O and analytics pipelines on high end machines (Fig. 1). SODA permits developers to embed analytics tasks into a componentized, dynamically managed execution and messaging framework, called a *workstation.* Such workstations have well defined inputs and outputs [11], can be parallel (MPI or threads), and may exhibit inter-workstation dependencies. Entire I/O pipelines can be constructed by chaining workstations along their I/O paths.

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

978-1-4673-9325-6/15 $31.00 © 2015 IEEE
DOI 10.1109/eScience.2015.59

techniques that guarantee control and orchestration actions taken by SODA do not place components into inconsistent states [12]. For example, SODA can prevent resource use until a different workstation has fully relinquished the resource. Such requirements become important as I/O pipelines scale geographically [13] as network partitions or data center outages can render parts of the pipeline inoperable.

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

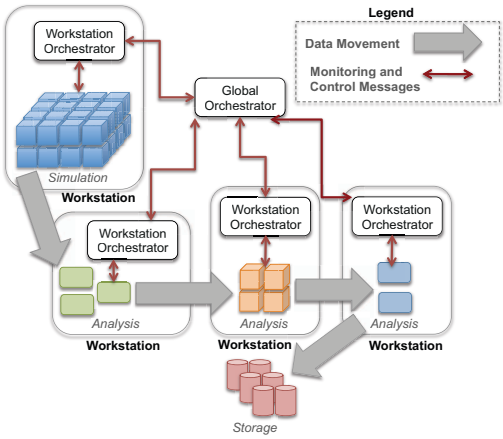


Figure 1. High-level view of the SODA framework.

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

Using two high end applications, the LAMMPS [14] molecular dynamics and the GTS [15] fusion simulations, along with different sets of analytics pipelines (SmartPointer [16] and a wave-space analysis code, respectively), we evaluate SODA with SLAs that include: (1) *bottleneck reduction* - a global performance-driven SLA that implements "elastic workstations" to remediate detected I/O pipeline bottlenecks; (2) *data reactive* - a workstation-level data-centric policy that changes component behavior based on data feature detection; and (3) *fault recovery* - a set of policies to handle an unexpected component departure such as analysis codes on an end-user device (e.g, a laptop). Experimental evaluations show that active, SODA-based management can: (1) respond to runtime dynamics at different stack levels; (2) create and enforce SLAs at multiple granularities in an I/O pipeline; and (3) operate at large scales with low overheads.

SODA constitutes new functionality in the scientific data management domain. Current I/O staging technologies do not offer support for dynamically managing end-to-end properties of tightly coupled analytics running with high end codes. For instance, earlier data staging work runs statically profiled analysis routines in configurations sized for worst case data volumes and processing needs [17]. Similarly, our recent supercomputer simluation "in-situ" analytics work [3] schedules and manages only the analytics actions taking place on individual compute nodes without concern for the I/O pipeline end-to-end properties affected by such nodes.

The contributions of this work are as follows: (1) we have specified a model for structuring analytics codes that allows them to be flexibly orchestrated at run time; (2) we present a programmatic API that allows for scientist-driven creation of SLAs; (3) an evaluation of three sets of representative SLAs using two real science simulations and associated analytics pipelines; and (4) to our best knowledge, we are the first to offer this kind of functionality under the constraints and requirements for typical science applications running on leadership class machines.

II. Related Work

Previous work on datacenter management and for "big data" systems uses techniques like elasticity and replication to provide scalability and fault tolerance [18], [19], [20], [21], but do not address directly the end-to-end behaviors and resource restrictions of the parallel analytics pipelines SODA manages. Specifically, with the SODA model, we can realize the diverse orchestration semantics needed for such pipelines expressed with SLAs and drive orchestration actions that implement the limited types of elasticity permitted by the HPC machine, the degree of reactivity needed for effective pipeline use, and the desired end-to-end behaviors, such as throughput or latency.

While SODA may appear like a limited hypervisor, it is not concerned with node-level partitioning and running multiple entities with performance and security isolation. SODA workstations are more akin to "resource islands" explored for high end multicore processors [22]. They differ in the explicit orchestration policies and actions specification and in enabling custom and application-specific methods for managing analytics pipelines.

Other HPC-centric work on managing analytics and visualization pipelines [23] provides adaptation policies at different stack layers (cross-layer adaptation) targeting an adaptive mesh refinement (AMR) code. It focuses on specific policies at different layers, to ensure minimal time to solution, whereas our work investigates the mechanics and abstractions of management that would be suitable for analytics pipelines; the policies discussed in [23] are examples of additional policies suitable for implementation with the SODA framework.

Initial results [24] demonstrate some of the concepts discussed here, but the work presented in this paper (1) extends upon the model and orchestration constructs, (2) explores a

wider variety of use cases, including an understanding of how state and metadata are managed (i.e., quality of data and fault recovery), (3) describes how SLAs are defined and how policies are constructed to enforce them, and (4) extends the concepts to pipeline that span multiple machines by leveraging the Flexpath [25] staging solution operating across a variety of interconnects. Our earlier solution was implemented with the Cray Portals API [26] and only operated on high end machines.

Our earlier work on "service augmentation" [27] demonstrates the utility of attaching Quality of Service (QoS) management actions to I/O pipelines and shows that SODA principles can be applied to other data staging or streaming infrastructures and systems, including DataSpaces [5] and Glean [6], both of which use componentized approaches.

III. SODA Framework

SODA is a set of run-time abstractions for dynamically orchestrating science applications and their associated analytics executables. Analytics executables are encapsulated in *workstations* that are connected along their I/O paths to form an *I/O pipeline*. Orchestration is conducted through an orchestration hierarchy and is guided by a flexible event-driven monitoring and control infrastructure. Fig. 1 depicts SODA's conceptual model.

A. Assumptions and Desired Properties

The SODA approach rests on assumptions that hold true for many large-scale scientific applications and their associated online analytics pipelines.

- **Functional Dependencies.** Analytics codes expect to ingest data matching specific formats and layouts. These analytics functions may need to transform data to meet algorithmic correctness and/or to export an analysis function's discoveries into the data itself. Given these dependencies in the data-plane, functions in an analytics pipeline may require in-order operation.
- **Heterogeneous Codes.** Analytics can have heterogeneous architectures and have a wide range of execution models, fault tolerance, and scaling characteristics.
- **Stringent Resource Constraints.** Resources are typically assigned to the compute job statically. Analysis codes are given "spare" resources, i.e., spare CPU cycles on simulation nodes [3], [4], reserved staging nodes [28], [5], or those on smaller, auxiliary clusters perhaps in different physical locations. Analytics pipelines, therefore, must operate with these limited resources, without interfering with the simulations and their output actions including by delaying simulation completion or adversely affecting other jobs running on the same platforms.

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

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

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

Orchestration on one component can jeopardize the execution of other components. For instance, consider trading resources between two analytics components when recovering from some detected bottleneck. A failure can occur if one component, using incorrect resource state data, attempts to use a resource that has not been fully relinquished by another component. Therefore: *(4) orchestration operations must be reliable and be resilient to failure.*

B. Conceptual Model

1) Workstations: A workstation, depicted in Fig. 2 allows analytics tasks to be embedded into a dynamically managed messaging and execution framework. The workstation's I/O interfaces are similar in concept to those used in modern Service Oriented Architectures (SOA). The workstation is comprised of a set of *active replicas* and a *workstation orchestrator* overseeing its execution.

Active Replicas. Unlike the replication techniques used in fault tolerant systems where replicas have identical internal states[29], active replicas in workstations are key to obtaining scalability: with traditional replication, each replica performs redundant computations on the same data items whereas active replicas perform their computations on different epochs of data assigned to them. For the use-case discussed in Section V-A, data is assigned to active replicas in a round-robin fashion, but additional communication patterns can be supported. Using active replicas, a workstation orchestrator can increase its degree of parallelism by spawning a new replica. While this is similar to how

Map-Reduce jobs scale, note that an individual data epoch may only be able to be processed by a fixed process count and that scalability comes from overlapping processing of different epochs.

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

2) Orchestration Constructs: Hierarchical orchestration affords three primary benefits. First, such hierarchies can be scaled with ease [30]. Second, distinct per-workstation orchestrators can offer customized management routines and separate their local, per-component management states from global state about entire I/O pipelines. Third, the hierarchy helps define authority. A global orchestrator is responsible for operations that re-organize entire pipelines. Workstation orchestrators are responsible for operations affecting only their components and resources and respond to management invocations from higher-level (global) orchestrators.

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

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

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

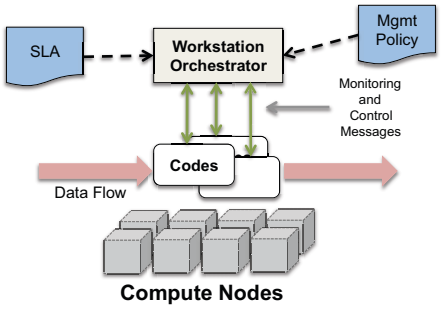


Figure 2. Workstation abstraction.

IV. Implementation

A. Workstation

1) Active Replicas: The implementation of SODA workstations leverages the widely used ADIOS read and write interfaces [11]. Using these interfaces, analytics codes can specify their data requirements and establish communication via a virtual file name serving as a named communication channel. To accommodate orchestration at runtime, the Flexpath [25] ADIOS transport, which allows for online analytics routines to exchange data, has been extended to accept and process management messages and state-change notifications from the replicas' designated orchestrator. We also modify the ADIOS interface to expose a communicator analytics applications can use to interact directly with the workstation orchestrator.

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

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

B. Orchestration Interface

The basic primitives listed in Section III-B2 are exposed as a C interface. Developers use this interface to create custom orchestrators if needed. SODA ships with some default implementations to automate elasiticity and recovery from a failed replica. To meet an SLA at a global orchestrator, the orchestrator can receive monitoring information as events and carries out chained primitives to perform actions like resource trading. When invoked, an orchestration primitive triggers a set of transactional protocols that indicate a participant's progress and distribute any state changes.

C. SODA Information Bus

Monitoring, control, and state change messages are delivered via the *SODA Information Bus*, or SIB, implemented using the EVPath [31] event-driven messaging library.

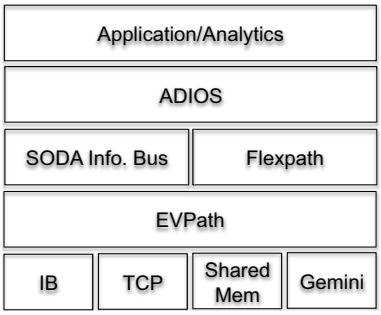


Figure 3. SODA software architecture.

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

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

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

D. Fault Detection and Recovery

The current implementation detects faults in two ways. The first uses application-level progress indicators delivered via periodic heartbeat messages from an application replica to its workstation-level orchestrator. The second allows the orchestrator to receive a notification from the kernel when the connection between an orchestrator and a replica has been broken. Method 1 does not rely on a specific messaging technology (e.g., sockets) and can work for a variety of underlying network interconnects with the disadvantage that the orchestrator must propagate failure notifications through the SIB to interested parties. Method 2 allows for any component interacting with it (orchestrators or other replicas in the pipeline) to receive the notification without waiting for failure alerts to propagate through the SIB. Both methods are explored in our current investigation because they are familiar to end-users and have well-understood characteristics. Future work will explore more robust fault detection [32], [33] and diagnostic [30] mechanisms.

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

V. Experimental Evaluation

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

The LAMMPS molecular dynamics simulation and the SmartPointer analysis toolkit serve as our application drivers for Titan. We construct two policies to demonstrate the benefit of the SODA approach and to assess the active management overheads. We run the GTS fusion simulation on Maquis and execute the spectral analysis (FFT) code on a machine at a remote location thereby allowing us to test the system's behavior when the pipeline is geographically distributed. We cannot conduct geographic experiments on Titan as its security policies and firewall settings prevent this.

A. Application Drivers

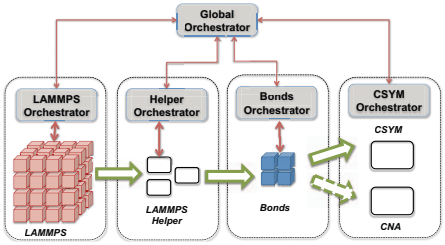


Figure 4. I/O Pipeline for LAMMPS with SODA

1) LAMMPS and SmartPointer: Figure 4 depicts the I/O pipeline constructed for the LAMMPS (Large Scale Atomic/Molecular Massively Parallel Simulator) [14] 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 (e.g., atom types and positions) with the size of this data ranging fom megabytes to terabytes depending on the science being investigated.

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. SmartPointer detects and categorizes the region geometry around the initial break and implements functions determining where and when plastic deformation occurs. We summarize the SmartPointer codes in the list below with additional detail found in [25], [24], [16]:

- *Lammps Helper*: parallel MPI code that aggregates and filters 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 has occurred. CNA is extremely compute-intensive ($O(n^3)$) and as such it should only execute when a crack has been detected.

2) GTS and FFT Analysis Code: As an alternative application example, to demonstrate the more general utility of SODA, we also evaluate our framework with GTS [15], a plasma fusion simulation with an implementation that exploits coarse grained process level parallelism using MPI and more fine-grained thread-level parallelism using OpenMP. This "particle in cell" code has different output frequencies for both particles and mesh-level statistics. To examine the dynamics involved, in particular dangerous transient effects that might damage a real reactor vessel, it is useful to dynamically evaluate and characterize particular trends on the inner and outer plasma edges. Unlike the LAMMPS case, these transients are not as algorithmically identifiable. Secondary analysis methods are used to infer their existence and then much more detailed inspection involving direct interaction with the physicists is used to further the investigation. The GTS analytics pipeline is a spectral code based on the AMD Core Math Libraries implementation of FFT that ingests the phi and Z-ion output arrays from the simulation.

B. Management Policies

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

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

Data-centric (Local): requires application introspection into the data based on the CSYM and CNA components. In contrast to the first policy, the analysis functions report the metric of interest (CSYM detects a crack) and the orchestration actions (kill CSYM and run CNA) are triggered by the workstation-level orchestrator. This policy ensures data quality via correct execution of pipeline analysis functions.

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

C. Quality of Data Policy and Microbenchmarks

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

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

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

Table I
INCREASE COMMAND PROTOCOL OVERHEAD

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

Table II
DATA-CENTRIC COMMAND PROTOCOL OVERHEAD

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

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

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

D. Throughput Measurements: QoS Policy

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

	LAMMPS	Helper	Bonds	CSYM
Fig 6(a)	8192	64	256 to 768	1
Fig 6(b)	4096	32	128 to 384	1
Fig 6(c)	2048	16	64 to 192	1

Table III
CORE COUNTS FOR THROUGHPUT EXPERIMENTS

Fig. 5 shows the baseline, unmanaged execution, for a LAMMPS simulation running on 8192 cores and a pipeline comprised of 64 Lammps Helper cores, 256 Bonds cores, and 1 CSYM core. The graph shows that as the output queue for Lammps Helper fills up, LAMMPS' throughput drops significantly. This is because it has to block on its output actions that must wait on queue space to free up. LAMMPS' throughput converges to that of Bonds, the slow component, effectively dropping end-to-end throughput to a third of the ideal target.

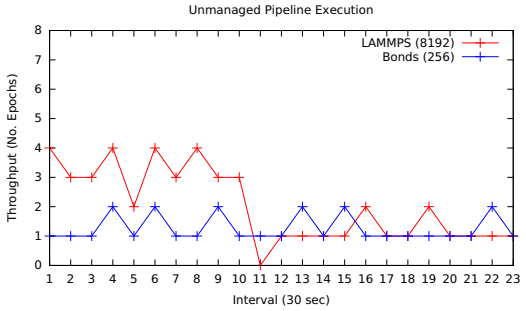
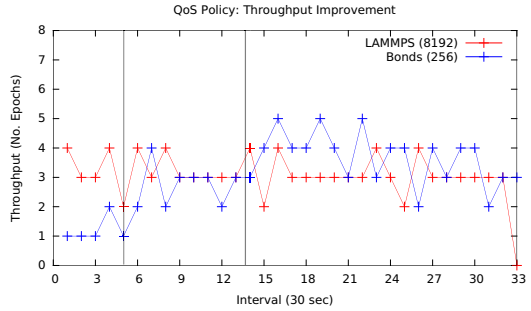


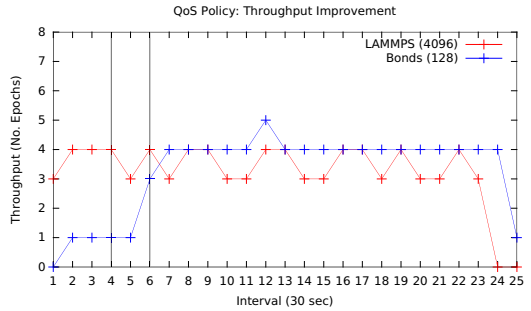
Figure 5. Throughput degradation for unmanaged pipeline.

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

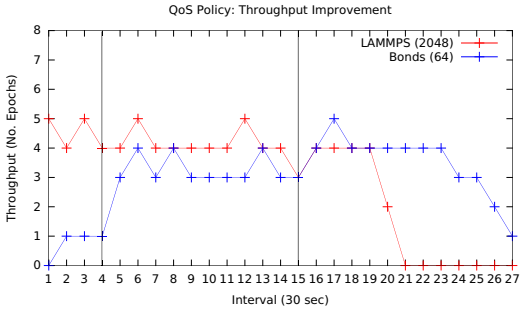
Figure 6(b) shows a similar result, where after three increases, Bonds maintains a slightly higher throughput than the LAMMPS simulation. However, speedup is insufficient to fully drain the queue in Lammps Helper, so the Bonds code executes somewhat longer. We see a similar phenomenon in Fig. 6(c), where the global orchestrator does not increase the Bonds workstation by an additional replica



(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.

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. 7 displays the changing queue length, the metric on which we base throughput management, for an experiment with the same setup as in Fig. 6(a). This represents the maximum queue length in the Lammps Helper workstation's output queue for the Bonds workstation. Here, the *x* axis represents the output epoch, and the *y* axis represents the max queue count when that output epoch is inserted into a queue. As is evident, the stated management policy is having the desired effect on its metric of interest.

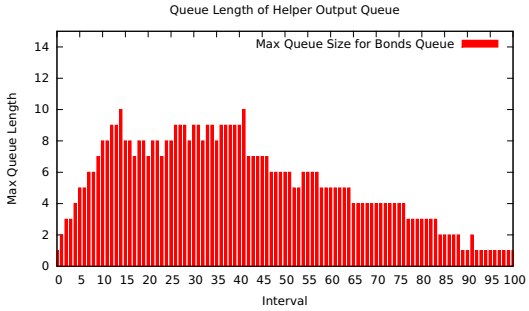
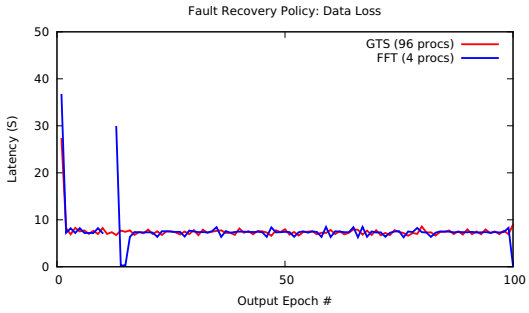
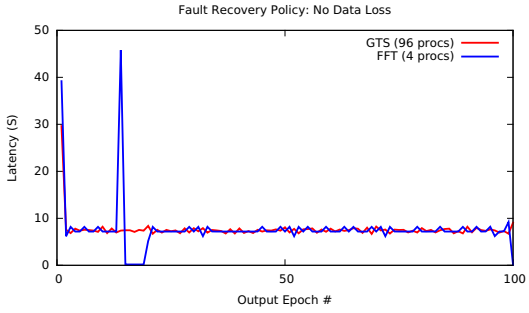


Figure 7. Change in max queue length for Helper Workstation.

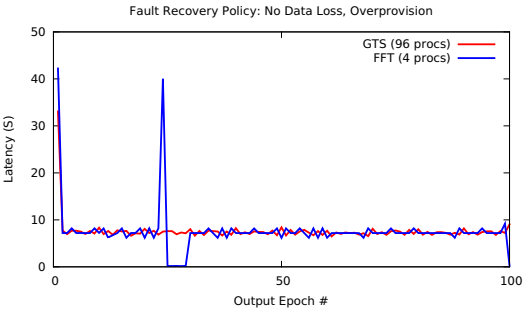
E. Fault Recovery Policy



(a) Fault Policy: Data Loss



(b) Fault Policy: No Data Loss



(c) Fault Policy: Overprovisioning

Figure 8. Failure recovery policy affect on latency

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

Fig. 8 displays the changes in workstation latency for three different fault-recovery mechanisms. The *x*-axis represents the epoch number for a workstation, and the *y*-axis represents the length of time between a step and the previous step. The first time step for each has a high latency, since we use the application start time as the base.

The first graph, Fig. 8(a), shows the workstation latency when recovering from a fault, but allowing for data loss, which is represented by the discontinuity for the FFT line. This has the lowest latency across all three because the previous (in other words, the older) time steps are simply dropped. Allowing for dropped epochs of data becomes more even more beneficial with configurations where it is infeasible, in terms of memory requirements, to buffer multiple timesteps of data.

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

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

F. Discussion

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

The performance results shown above demonstrate the superiority of managed vs. unmanaged I/O, guided by simple policies realized with low cost management structures. While able to scale to the high end machines currently available to our research, the current management policies implemented for SODA assume each workstation running on its own dedicated resources, separate from those used by the application. Management actions that involve scheduling or resource sharing [3] remain part of our future work.

VI. Conclusions and Future Work

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

Our future work will address two different dimensions of SODA-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 an science analytics pipeline 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 [3] and giving rise to concerns with performance isolation offered by virtualization technologies.

VII. 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

References

[1] S. Klasky and et al, "Grid -based parallel data streaming implemented for the gyrokinetic toroidal code," in *Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*, ser. SC '03. New York, NY, USA: ACM, pp. 24–.

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

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

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

[5] C. Docan and et. al, "Dataspaces: An interaction and coordination framework for coupled simulation workflows," ser. HPDC '10. ACM, 2010, pp. 25–36.

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

[7] P.-S. Koutsourelakis, "Accurate Uncertainty Quantification Using Inaccurate Computational Models," *SIAM J. Scientific Computing*, vol. 31, no. 5, pp. 3274–3300, 2009.

[8] F. Zhang and et. al, "Xpressspace: a programming framework for coupling partitioned global address space simulation codes," *CCPE*, vol. 26, no. 3, pp. 644–661, 2014.

[9] G. Wang and T. S. E. Ng, "The impact of virtualization on network performance of amazon EC2 data center," in *INFOCOM*, 2010, pp. 1163–1171.

[10] Y. Kanemasa and et. al, "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*, 2013, pp. 136–143.

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

[12] J. Lofstead, J. Dayal, K. Schwan, and R. Oldfield, "D2t: Doubly distributed transactions for high performance and distributed computing," in *Cluster*, Sept 2012, pp. 90–98.

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

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

[15] W. X. Wang and et. al, "Gyro-Kinetic simulation of global turbulent transport properties in tokamak experiments," *Physics of Plasmas*, vol. 13, no. 9, p. 092505, 2006.

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

[17] F. Zheng and et. al, "Predata: preparatory data analytics on peta-scale machines," in *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, April 2010, pp. 1–12.

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

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

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

[21] Storm: Distributed and fault-tolerant realtime computation. [Online]. Available: https://storm.incubator.apache.org/

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

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

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

[25] J. Dayal, "Flexpath: Type-based publish/subscribe system for large-scale science analytics," in *CCGrid '14*, pp. 246–255.

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

[27] M. Wolf and et. al, "Service Augmentation for High End Interactive Data Services," in *CLUSTER*, 2005, pp. 1–11.

[28] H. Abbasi and et. al, "DataStager: scalable data staging services for petascale applications," *Cluster Computing*, vol. 13, pp. 277–290, 2010.

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

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

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

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

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