NORNS: Extending Slurm to Support Data-Driven Workflows through Asynchronous Data Staging

Citation for published version:
Miranda, A, Jackson, W, Tocci, T, Panourgias, I & Nou, R 2019, NORNS: Extending Slurm to Support Data-Driven Workflows through Asynchronous Data Staging. in *2019 IEEE International Conference on Cluster Computing (CLUSTER).* Institute of Electrical and Electronics Engineers (IEEE), pp. 1-12, IEEE Cluster 2019, Albuquerque, New Mexico, United States, 23/09/19. https://doi.org/10.1109/CLUSTER.2019.8891014

Digital Object Identifier (DOI):
10.1109/CLUSTER.2019.8891014

Link:
Link to publication record in Edinburgh Research Explorer

Document Version:
Peer reviewed version

Published In:
2019 IEEE International Conference on Cluster Computing (CLUSTER)

NORNS: Extending Slurm to Support Data-Driven Workflows through Asynchronous Data Staging

Alberto Miranda*, Adrian Jackson†, Tommaso Tocci*, Iakovos Panourgias†, Ramon Nou*
*Barcelona Supercomputing Center, †EPCC, The University of Edinburgh
{alberto.miranda, ramon.nou, tommaso.tocci}@bsc.es, {a.jackson, i.panourgias}@epcc.ed.ac.uk

***Abstract*—As HPC systems move into the Exascale era, parallel file systems are struggling to keep up with the I/O requirements from data-intensive problems. While the inclusion of burst buffers has helped to alleviate this by improving I/O performance, it has also increased the complexity of the I/O hierarchy by adding additional storage layers each with its own semantics. This forces users to explicitly manage data movement between the different storage layers, which, coupled with the lack of interfaces to communicate data dependencies between jobs in a data-driven workflow, prevents resource schedulers from optimizing these transfers to benefit the cluster's overall performance. This paper proposes several extensions to job schedulers, prototyped using the Slurm scheduling system, to enable users to appropriately express the data dependencies between the different phases in their processing workflows. It also introduces a new service for asynchronous data staging called NORNS that coordinates with the job scheduler to orchestrate data transfers to achieve better resource utilization. Our evaluation shows that a workflow-aware Slurm exploits node-local storage more effectively, reducing the filesystem I/O contention and improving job running times.**

***Index Terms*—Scientific Workflows, Burst Buffers, High-Performance Computing, Data Staging, In Situ Processing**

I. Introduction

As HPC systems become capable of computations in the order of hundreds of petaFLOPs [1], researchers are increasingly turning their attention to more complex problems which require the large-scale analysis of experimental and observational data (EOD), such as the computational analysis of ITER reactor designs [2] or the simulation, filtering, and evaluation of large-scale experiments such as the Compact Muon Solenoid at the Large Hadron Collider [3][4]. What separates these *data-intensive problems* from traditional, compute-bound large-scale simulations is that, even though they exhibit comparable computational needs, they also incur significant data requirements. Thus, while much of the I/O volume in traditional HPC applications comes from checkpoint files [5], which are written once and almost never read back, data-intensive applications generate and consume data at much larger volumes and/or rates[1]. What's more, in many cases these problems are run as a composition of separate tasks that are executed as a *scientific workflow* in the context of a large parallel job [8], with each task representing a separate phase in a complex model that may depend on data generated by previous tasks. This requires communicating large amounts of data between tasks, which is typically accomplished by means of the storage subsystem. Thus, even if modern HPC clusters are capable of running many such applications at the same time to maximize the use of supercomputing resources, severe I/O performance degradation is often observed due to uncoordinated, competing accesses to shared storage resources.

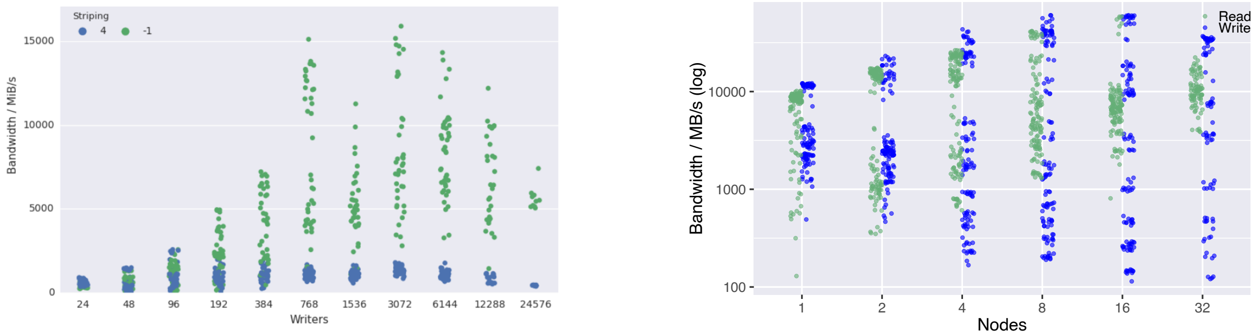
To mitigate this phenomenon, burst buffers such as Cray Datawarp [9] or DDN IME [10] are being increasingly included into the HPC storage architecture to reduce I/O contention. By writing data to these burst buffers rather than to the cluster's parallel file system (PFS), processes running on compute nodes can reduce the time required for reading and/or writing data to persistent storage, even if this storage is not the data's ultimate destination. For example, the Cori system at NERSC and the Trinity supercomputer at LANL rely on Cray Datawarp to accelerate I/O [11][12]. However, appliances typically consist of dedicated storage hardware deployed onto separate I/O nodes and available as an I/O resource that is external to the compute nodes in the same way a traditional parallel filesystem is accessed. This means that burst buffers require correct sizing to ensure they can adequately handle the volume of I/O the system may produce, exactly the same as parallel filesystems. The increasing availability of low-latency storage devices and interfaces such as Non-Volatile Memory Express (NVMe) and Intel® Optane™ [13] has favored new architecture designs where such technologies are included into compute nodes. For instance, the Summit [14], Sierra [15], and MareNostrum IV [16] supercomputers include NVMe devices as *node-local burst buffers*. Since these devices often exhibit faster performance than NAND Flash Solid-State Drives (SSDs), their inclusion as node-local storage effectively adds a new high-performance staging layer where processes can efficiently store files, thus moving data closer to the computing resources that require it.

Nevertheless, as multiple layers of storage are added to the HPC I/O architecture, the complexity of transferring data between these layers also increases [17]. Unfortunately, while computing and network resources can be shared and managed effectively by state-of-the-art job schedulers, storage resources are still mostly considered as black boxes by these infrastruc-

This work was partially supported by the Spanish Ministry of Science and Innovation under the TIN2015–65316 grant, the Generalitat de Catalunya under contract 2014–SGR–1051, as well as the European Union's Horizon 2020 Research and Innovation Programme, under Grant Agreement no. 671951 (NEXTGenIO). Source code for NORNS is available at https://github.com/NGIOproject/NORNS Source code for Slurm extensions is available at https://github.com/NGIOproject/Slurm.

[1] The LHC data output is expected to grow to 150 PB/year + 600 PB/year of derived data by 2025, while the Large Synoptic Survey Telescope will generate 3.2 Gigapixel images every 20 seconds [6][7].

978-1-7281-4734-5/19/$31.00 ©2019 IEEE



(a) ARCHER

(b) MareNostrum IV

Fig. 1. Impact of cross-application interference in I/O performance

ture services [18]. While there has been increasing interest in HPC to use burst buffers to optimize the I/O path of data-driven workflows through autonomous, asynchronous data staging [19][20][21], these research efforts have not considered I/O as a first class entity in resource scheduling decisions. Thus, we argue that the integration of application I/O needs with scheduling and resource managers is critical to effectively use and manage a hierarchical storage stack that can include as many layers as NVRAM, node-local burst buffers, shared burst buffers, parallel file system, campaign storage, and archival storage. Improved coordination between the storage system and the job scheduler can reduce I/O contention resulting in reduced job runtimes and improved system efficiency.

In this paper, we propose a new infrastructure service for HPC clusters called NORNS, that coordinates with the job scheduler to orchestrate asynchronous data transfers between the available storage layers. NORNS provides facilities to system administrators to expose the cluster's storage architecture to job schedulers and applications, and also offers interfaces for creating and monitoring asynchronous data transfers between these local and remote layers. We also propose extensions to the Slurm job scheduler [22] so that users can define jobs as processing phases of larger I/O-driven workflows, and also provide information about how data resources should propagate through the storage layers in each workflow phase. Using this information, Slurm can rely on the NORNS API to create appropriate I/O tasks to stage-in/stage-out data from/to the PFS when the workflow starts/ends, keep persistent data on node-local storage to feed upcoming phases or move data directly between compute nodes to match future job schedules. NORNS, thus, provides a framework for the development of data-aware scheduling algorithms to arbitrate these transfers, so that it is possible to maximize transfer throughput and minimize interferences with application I/O.

This work makes the following contributions: (1) we present the Slurm extensions for data-driven workflow support in Section III; (2) we introduce the design and implementation of the NORNS service in Section IV-A and Section IV-B, respectively; (3) we discuss the NORNS user and administrative APIs in Section IV-C; and (4) we finalize in Section V by evaluating the benefits for HPC clusters of using data-driven workflows as a central job scheduling component.

II. Background and Motivation

As we move towards the Exascale era, the I/O performance gap widens [23][24]. While large scale environments have traditionally relied on remote parallel file systems such as Lustre [25] or GPFS [26], most of these file systems are facing significant challenges in performance and scalability due to the massive amounts of data produced by EOD-driven applications [27]. Figure 1 shows the impact of this interference on the I/O performance of the ARCHER [28] and MareNostrum IV [16] supercomputers.

ARCHER is a Cray XC30 system with 24 cores (two Intel Xeon 2.7 GHz, 12-core E5-2697v2 processors) and 64 GB of DDR3 memory per node (128 GB on a small number of large memory nodes). Its 4,920 nodes are connected by the Cray Aries network, with three Lustre filesystems. The Lustre filesystem used for benchmarks presented in this paper have 12 Object Storage Services (OSSs), each with 4 Object Storage Targets (OSTs) containing 10 disk drives (4TB SEAGATE ST4000NM0023). This gives 40 disks per OST (configured in RAID6 mode) and 480 disks for the filesystem. There is a single Metadata Service for the filesystem.

For ARCHER we repeatedly ran an MPI-I/O benchmark using the same number of process/nodes once a day for an extended period of time. Each benchmark was run using two different Lustre striping options (either the default stripe, which used 4 OSTs, or using all the OSTs in the filesystem). The benchmark writes to a single file across all processes using collective MPI-I/O functions. The maximum achieved bandwidth for writing 100MB of data per writer is presented in Figure 1a. This figure shows that, when using sufficient numbers of writing processes it is possible to achieve close to the 20GB/s of theoretical write performance the filesystem provides. However, this can only be achieved when using the full Lustre striping (i.e. all the filesystem OSTs), and even in that circumstance we can see a four fold difference in achieved bandwidth between the fastest (around 16GB/s) and slowest (under 3GB/s) results collected for a given number of writers. The only difference between any one data point using the same number of writers is the amount of other network communication and filesystem traffic occurring at the same time as the benchmark is being undertaken.

MareNostrum IV has 48 racks with 3,456 Lenovo ThinkSystem SD530 compute nodes. Each node has two Intel Xeon Platinum 8160 24C chips, each with 24 processors at 2.1GHz, amounting to a total of 165,888 processors and a main memory of 390 TB. The interconnection network is a 100Gb Intel Omni-Path Full-Fat Tree and its 14PB of storage capacity are offered by IBM's GPFS. For MareNostrum IV (Fig. 1b), we used the IOR benchmark [29] to measure the I/O performance variability observed by applications depending on the PFS load. To measure variability, we ran 25 independent repetitions of the same benchmark during one week, each in a different node allocation and on a different time frame. The benchmarks ran co-located with the normal HPC workload, and were configured to create independent files per core in each compute node, using 24 out of 48 cores. To avoid cache effects, file sizes were chosen to be large enough to fill the node's memory. The benchmark then proceeded to read/write from/to the created files, using a transfer size equal to the file system's. As with ARCHER, the results show extremely high variability in GPFS' I/O performance, with measured bandwidths often diverging by orders of magnitude. This extreme variation present in both machines, is due to a phenomenon known as *cross-application interference* which happens when multiple applications access a shared resource (e.g. the PFS) in an uncoordinated manner. This I/O congestion is such a common occurrence in most HPC sites [30][31][32], that some studies suggest that this so-called *I/O bottleneck* could be one of the main problems for Exascale machines [33][34][35].

While the widespread inclusion of SSDs, NVMe devices and shared burst buffer nodes in HPC storage has alleviated this issue by improving I/O performance, it has done so at the cost of increasing complexity in the I/O hierarchy. This introduces multiple challenges in the space of possible configurations and complicated interactions. First, burst buffers (shared or node-local) can be used (at least) as temporary storage for intermediate data, to cache PFS data, and as a medium to share data between workflow phases. Second, users now need to explicitly manage the placement and movement of application data between the different storage layers, a clearly sub-optimal solution since users obviously lack information about the global state of the system as well as the best moment to perform such movements. Moreover, it also forces users (i.e. scientists and researchers) to spend some time learning the best way to use these technologies in their applications, an effort better spent in their scientific problems. Third, opportunities for global I/O optimization are lost by not communicating application needs to the infrastructure services in charge of resource allocation, which could exploit this information to produce better scheduling policies to reduce contention. Thus, any interfaces that do not expose any information about the storage hierarchy to applications, and relying solely on the OS and hardware to transparently manage the storage stack will lead to sub-optimal performance.

Unfortunately, the currently available interfaces between users and resource managers do not make it possible to convey *data dependencies* between jobs to model the different phases in a workflow. For example, if a job A generates data that should be fed into a job B, there is no way for users to express this dependency, nor to influence the job scheduling process so that A's output is kept in burst buffers until B starts. Worse yet, since the I/O stack remains a black box for today's job schedulers, job A's output could end up being synchronized to the cluster's PFS and, at some point in the near future, staged back into the new node allocation for job B, which might end up including some of the original nodes reserved for job A.

Thus, capturing application I/O requirements and appropriately exposing the layers in the new storage architecture offers advantages for the overall I/O performance of the HPC cluster. More specifically, appropriate usage of NVM-based node-local burst buffers would provide several benefits. First, restricting applications to rely on node-local storage is an effective way to prevent them from arbitrarily causing contention to the PFS. By leveraging node-local storage, the cluster's PFS would only need to be accessed during two well-controlled situations: (1) to copy input data into node-local storage; and (2) to persist output data to the PFS for long-term storage. Second, replacing normal application I/O by stage-in/processing/stage-out workflows would allow for a more effective use of the PFS: since accesses to the PFS would be mostly restricted to staging phases, several staging phases could be scheduled to run concurrently while minimizing I/O contention. Moreover, from the point of view of the PFS, application I/O would transform from a stream of unrelated, random data accesses, to well-defined sequential read/write phases. Third, EOD-driven workflows could take advantage of high-density node-local NVM for data to be left *in situ* for the next workflow phase. Fourth, leveraging node-local storage hardware can allow application-observed I/O performance to scale with the number of compute nodes used, with larger parallel runs acquiring larger I/O bandwidth through the aggregation of the I/O hardware in the number of nodes used for the job.

Addressing these challenges motivates the design of the proposed scheduler extensions and the NORNS service. Through this new infrastructure service, we aim to provide a robust set of interfaces and data transfer primitives so that specialized scheduling algorithms can be developed to optimize I/O performance in the HPC cluster.

III. Slurm Extensions for Data-Driven Workflows

Computational simulation or data analytics and machine learning activities generally are not confined to using a single application on a single data set. Most users of compute resources will run a range of applications, scripts, and tools, to process, produce and analyze data. As such, they will have a workflow that they use to undertake their work, composed of a range of applications. Unfortunately, most mainstream job schedulers do not support rich workflow functionality. Even though some schedulers support dependencies, however, dependant jobs are generally treated as new tasks by scheduling algorithms, ignoring the history of previous dependencies in the workflow. Moreover, this approach does not support sharing data on compute nodes between stages in a workflow,

```
#NORNS stage_in origin destination mapping
#NORNS stage_out origin destination mapping
#NORNS persist operation location user
```

Listing 1. Scheduler options and parameters for specifying job data dependencies and operations

which is one of the key advantages of node-local persistent storage hardware. Thus, to enable users to efficiently share data between workflow phases utilizing node-local storage without requiring all compute nodes to be reserved for the full duration of the workflow[2], we propose extensions to fully integrate user workflows with the scheduler and its scheduling algorithms.

We have developed prototype workflow functionality that allows the scheduling algorithms of the Slurm job scheduler to consider all jobs that are part of a workflow as a unit. Each intermediate job gets updated priorities and resource allocations as the different phases progress, even though an intermediate job might not be able to start due to dependencies. Moreover, a dependant job cannot start before all its dependencies are satisfied. The workflow is supported through new options passed to the scheduler by the user, as part of their job submission script, which specify if a job is the start of a workflow (`workflow-start`), the end of a workflow (`workflow-end`), or has a dependency with another job in the workflow (`workflow-prior-dependency ID`). Each workflow is assigned a unique *Workflow ID* enabling users to be able to enquire about the overall status of a workflow and obtain a list of all jobs and their status (running, pending, etc.). If a workflow job fails; then all subsequent jobs are cancelled and users can react accordingly.

Workflow functionality in isolation, however, does not provide the ability to share data between workflow components. Such functionality requires further extensions to the job scheduler to enable each workflow component to specify what data should stay on node-local persistent storage, as well as which data is required for input to, or will be produced as output by, a given workflow component. To enable such functionality we added further batch system options to allow users to specify data input, output, and dependencies for a workflow component job. These consist of `stage_in`, `stage_out`, and `persist` options that the user can specify a given data source/sink with to enable data to be shared between workflow components. These options accept a range of parameters to specify the types and locations of data under consideration, as outlined in Listing 1.

Parameters `origin`, `destination`, and `location` must refer to pre-existing dataspaces or data resources (refer to Section IV-A for details), while `location` must be a node-local storage resource. Argument `user` must be a username in the form that the scheduler recognizes, and argument `operation` can be one of the following options:

- `store`: take a location and maintain it on a node-local storage resource
- `delete`: delete an existing persisted location from node-local storage resources
- `share`: add permission for user to access the persist location
- `unshare`: remove permission for user to access the persist location

The `mapping` argument is used to specify how data is mapped from shared to node-local resources and vice versa. For a single node Slurm job, mapping is not required as there may only be a single set of node-local resources to place data on, although mapping may be used to specify which specific node local resource data items are mapped to if there are multiple node-local resources (i.e. where there is a mount point per socket on a dual socket node).

Prior to the launch of an individual job, the scheduler will have sufficient information to trigger data movement for the nodes chosen for a given job to move any data required for that job (i.e `stage_in` operations). The scheduler uses calculations of average data transfer times and data sizes to decide when to trigger such movements prior to a job starting. When the job is ready to start, the scheduler will check the data has arrived. If the data is ready the compute part of the jobs is started. If the data is still being transferred the scheduler will wait until the transfer is complete or until a pre-configured timeout is encountered. If the timeout is reached or if there is a failure to obtain the data item specified, the scheduler will terminate the job and clean up all data already staged to nodes.

Similar operations are undertaken on a `stage_out` request, except this happens at the end of a job run. A mappings file can also be used for stage out processes, although this can simply be a single directory to copy all the data into. If a `stage_out` operation fails then the current approach is to leave the data on the node local resources for future `stage_out` operations to try and recover.

Whilst all the functionality we have discussed requires the batch scheduler to coordinate and provide user interfaces to enable efficient and effective use of the storage hardware environments we are targeting in this work, it does not necessarily make sense to implement all this functionality within the job scheduler. For flexibility, we decided to create a standalone service, the NORNS service, to provide the functionality to manage and move data on compute nodes, with interfaces defined to enable both user and system applications to interact with the service and request operations.

IV. The NORNS Service

NORNS is an infrastructure service whose main goals are to facilitate exploiting the cluster's storage architecture, and to coordinate with the scheduler to orchestrate data transfers between the different storage layers in an HPC cluster. Its design objectives are the following:

- To simplify the job scheduler's management of the increasingly heterogeneous I/O stack by offering interfaces for abstracting and controlling the different storage layers, as well as modelling and managing batch jobs/workflows.

[2] Which may be wasteful as workflows often have varying compute requirements for different parts of the workflow.

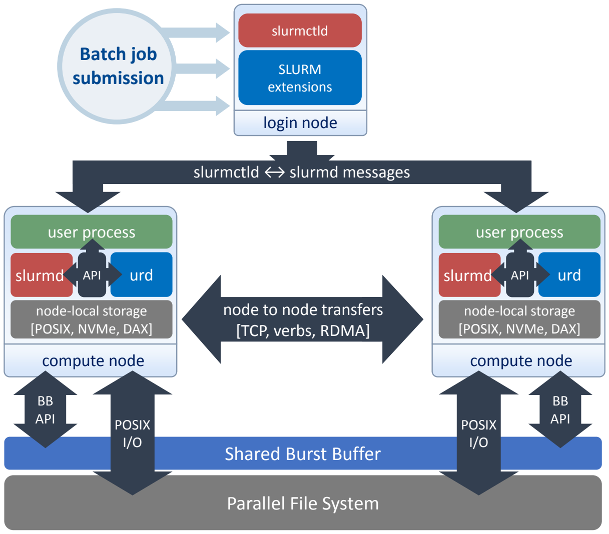


Fig. 2. NORNS service architecture and component interaction.

TABLE I
PRINCIPAL INTERFACES IN THE NORNS APIS

Control API (All functions share the `nornsctl_` prefix)	

Daemon management	`send_command(cmd, args),`
	`status(cmd, args)`

Dataspace management	`backend_init(flags, path),`
	`register_dataspace(DSID, backend),`
	`update_dataspace(DSID, backend),`
	`unregister_dataspace(DSID)`

Job management	`job_init(hosts, limits),`
	`register_job(JOBID, job),`
	`update_job(JOBID, job),`
	`unregister_job(JOBID)`

Process management	`proc_init(PID, UID, GID),`
	`add_process(JOBID, proc),`
	`remove_process(JOBID, proc),`

Task management	`resource_init(type, ...),`
	`iotask_init(type, input, output),`
	`submit(task), wait(task, timeout),`
	`error(task, stats)`

User API (All functions share the `norns_` prefix)	

Dataspaces & task management	`get_dataspace_info(dataspaces),`
	`resource_init(type, ...),`
	`iotask_init(type, input, output),`
	`submit(task), wait(task, timeout),`
	`error(task, stats)`

- To hide the complexity of each specific storage layer by providing a unified interface for transferring data, so that the job scheduler and end-users do not need to bother with the technical details to execute such transfers efficiently.
- To allow the job scheduler to start, monitor, and manage transfers between storage layers, so that it can control and account the data staging required to run a job.
- To allow parallel applications to start, monitor, and manage transfers between the different storage layers made available to a job by the scheduler, while restricting access to any layers not explicitly allowed by the scheduler.
- To execute data transfers as efficiently as possible, taking advantage of fast interconnects and native APIs where available, and minimizing contention to the PFS.

NORNS, hence, serves as the main component that validates, orchestrates, and executes the required transfers to run an already scheduled job. By executing these transfers asynchronously, NORNS allows the job scheduler to submit requests to transfer data without having to wait for it to actually be available at the intended destination. This frees the scheduler to continue scheduling oncoming job submissions.

A. Design and Architecture

Since its main goal is to keep track of any data staging required to run a job, the architecture of the NORNS service has been designed to be tightly coupled with job schedulers. As shown in Figure 2, the NORNS service is currently composed of the following components:

- A resource control daemon called `urd` which runs at each compute node. Its responsibilities are to coordinate with the job scheduler to manage and track the dataspaces defined for each batch job. It is also the component in charge of actually accepting, validating, executing, and monitoring all I/O tasks affecting this compute node both by the job scheduler and parallel applications.
- A control `nornsctl` API. This API offers administrative interfaces so that the job scheduler can control the `urd` daemon, query its state, define the appropriate dataspaces for each job, and submit/control I/O tasks.
- A user-level `norns` API. This API allows parallel applications running in the context of a batch job or workflow to query information about any dataspaces defined for them, and also offers interfaces so that they can submit and control I/O tasks between such dataspaces.

Thus, considering a practical example integrating with one such job scheduler, Slurm, when a batch job/workflow is submitted, `slurmctld`[3], through the extensions discussed on Section III, captures and internally registers both the storage layers affected by the job and its I/O requirements. Based on this information, Slurm schedules the next job to run, reserves a set of nodes for it and communicates with `slurmd`[4] to configure the nodes. Note that this implied adding additional messages to the usual communication between `slurmctld` and `slurmd`, which performs the actual calls to the `nornsctl` API. Any storage resources available to a job will be assigned an ID by `slurmctld` (e.g. "`lustre://`", "`nvme0://`", or "`pmdk0://`"), which will then contact the `slurmd` daemons in the nodes reserved for the job so that they register them in the local `urd` daemons through the control API. This creates an abstraction called *dataspace*[5] that hides from users the technical details of available storage tiers (e.g. local/shared burst buffers, PFS) and allows Slurm to account and limit how tiers are used by jobs by leveraging OS control facilities such as cgroups or chroot if the tier supports them.

[3] The principal Slurm control daemon users interact with.

[4] A Slurm daemon in charge of controlling each compute node.

[5] A portmanteau for data namespace.

When all dataspaces are defined, the job scheduler may satisfy the job's I/O requirements by creating and submitting several *I/O tasks* to execute data transfers between the available dataspaces. This typically means transferring data from either the shared PFS/burst buffer dataspace using specialized APIs (e.g. POSIX/MPI-IO, or the DataWarp API), or from a dataspace defined in a remote node using the cluster's interconnect (e.g. Intel®'s Omni-Path™, or Cray®'s Gemini/Aries™).

At this point, each `urd` daemon in the involved compute nodes will be responsible for asynchronously transferring the requested data, as well as monitoring the performance of such transfers in order to compute an E.T.A. for each task. This is done so that `slurmctld` can estimate how long a node may be "in use" by data transfers before a job starts and after a job completes, so that it can fine-tune its internal scheduling algorithms. Once the initial set of I/O tasks has completed, the job can be safely started using the data located in node-local storage. In order to use the new dataspaces, users only need to change their scripts to use the appropriate environment variables provided by Slurm when starting the job processes (e.g. `$LUSTRE`, `$PMDK0`, `$NVME0`), which correspond to the I/O requirements defined by the user when submitting the job. Alternatively, user code may also rely on the `norns` API to query information about the dataspaces exposed for the application. Note that the `norns` API may also be used by applications while the job is running, in order to schedule asynchronous data transfers between the dataspaces they have access to (e.g. to offload memory buffers to node-local storage for checkpointing). Once a job finally completes, Slurm may schedule additional I/O tasks before releasing compute nodes (e.g. output results may need to be transferred to the PFS or to another set of nodes for further processing). In this situation, NORNS E.T.A. tracking becomes even more important since it allows Slurm to determine at which time a compute node will re-enter the pool of free resources. Note as well that user transfers while the job was running may leave data in local dataspaces unbeknownst to Slurm. To account for this situation, Slurm can optionally prompt NORNS to "track" dataspaces so that it checks whether they are empty if a node release is attempted. If data remains in a tracked dataspace Slurm will be informed of the presence of a non-empty dataspace, which will allow it to take appropriate measures.

B. Implementation

For performance, the NORNS service has been written in C++. Figure 3 shows the internal components of the `urd` daemon and its interactions with client processes. When either Slurm (through `slurmd`) or an application needs to send a request, it only needs to invoke the desired functions exposed by the appropriate API, which capture the request parameters and forward them to the daemon. This message passing is done by sending messages serialized with Google's Protocol Buffers [36] through local `AF_UNIX` sockets. Note that since both APIs are built as standard ELF shared objects that any application can link to, any application may potentially invoke any of their functions. To prevent unauthorized use, two separate "control" and "user" sockets are created with differing file system permissions. In this way, if an API function is invoked, the function can verify whether the calling process actually has permission to access the underlying socket and deny the request if it does not. This allows administrators to restrict usage of the APIs using the kernel's security mechanisms, for instance by creating a `norns` group for any applications that can make administrative changes to the service, and a `norns-user` group for user applications allowed to use the NORNS service.[6]

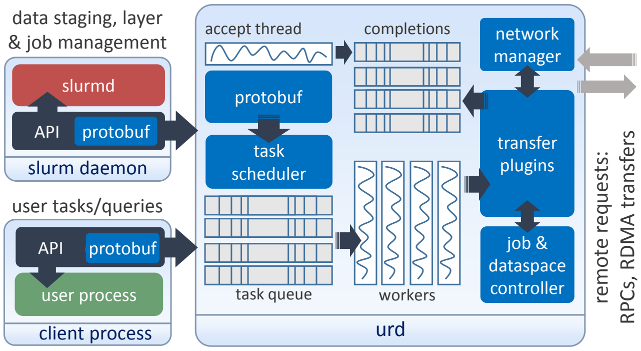


Fig. 3. Internal components of the urd daemon

When incoming requests from any API reach the `urd` daemon, they are processed by a dedicated *accept thread* that relies on the kernel's `epoll(7)` mechanism to implement an asynchronous event handling pattern for high availability. When a request is received, the accept thread deserializes it, creates a task descriptor for it and places into a *task queue* for pending requests. Note that task order in the queue is controlled by a *task scheduler* component, which arbitrates the order of the execution of I/O tasks depending on several metrics. FCFS is the default arbitration policy, but the component will be extended in the future to support other strategies.

I/O tasks are then extracted from the queue and executed by `urd`'s *worker threads*, which will rely on the information registered in the *job & dataspace controller* to validate the request, which implies checking that the calling process has access to the requested dataspaces and also that it has the appropriate file system permissions to access the requested resources. Note that, for flexibility, NORNS supports defining specific plugins to transfer data between a pair of resource types, which allows developers to write high performance data transfers based on the internals of each data resource. In this way, NORNS can be easily extended to efficiently support additional data resources as they become available. Thus, if the task is deemed as valid, the worker thread will select the appropriate *transfer plugin* for the task at hand and will initiate the transfer (Table II shows the currently supported plugins). Moreover, since some I/O tasks may require orchestrating data transfers between `urd` daemons running in separate compute nodes, the `urd` daemon also includes a *network manager* component that allows sending RPCs to remote `urd` daemons,

[6] Note that Slurm may add job processes to the `norns-users` group by using the `setgroups(2)` system call upon creation.

TABLE II
TRANSFER PLUGINS SUPPORTED BY NORNS

Plugin	Simplified Operation

Process memory ⇒ local path	`out=fallocate()+mmap();`
	`process_vm_readv(in, out);`

Memory buffer ⇒ remote path	*At initiator:*
	`tmp=fallocate()+mmap();`
	`process_vm_readv(in, tmp);`
	`send_to_target(tmp_info);`
	At target:
	`RDMA_PULL(tmp_info, out);`

Memory buffer ⇐ remote path	`in_info=query_target(in);`
	`tmp=fallocate()+mmap();`
	`RDMA_PULL(in_info, tmp);`
	`process_vm_writev(tmp,out);`

Local path ⇒ local path	`in_fd=open(in, RDONLY);`
	`out_fd=open(out, WRONLY);`
	`sendfile(in_fd, out_fd);`

Local path ⇒ remote path	*At initiator:*
	`in_info=mmap(in);`
	`send_to_target(in_info);`
	At target:
	`RDMA_PULL(tmp_info, out);`

Local path ⇐ remote path	`in_info=query_target(in);`
	`out=fallocate()+mmap();`
	`RDMA_PULL(in_info, out);`

and also supports RDMA-capable transfers if the interconnect fabric supports it. The implementation of this component relies on ANL's Mercury library [37], a C library for implementing RPCs and bulk data transfers that has been especially designed (and optimized) to be used in HPC clusters. Mercury includes a Network Abstraction (NA) layer which provides a minimal set of function calls that abstract from the underlying network fabric and that can be used to provide target address lookup, point-to-point messaging, remote memory access, and transfer progress/cancellation. Most interestingly for NORNS, Mercury's NA layer uses a plugin mechanism so that support for various network fabrics such as Ethernet, Omni-Path™, Gemini/Aries™, or InfiniBand™ can be easily selected at runtime, making them transparently available to NORNS.

Finally, upon a task's completion, the worker thread responsible for executing it will place a descriptor of its completion status into a *completion list*, so that clients can query it with the appropriate API interfaces.

C. APIs

Table I shows the principal interfaces exposed by both the `nornsctl` and `norns` APIs. The APIs are written in C, since its flexibility to being used from other programming languages simplifies creating bindings for other languages such as Fortran, C++, Java, or Python, commonly used in HPC. The `nornsctl` administrative API has been designed to allow job schedulers to control the `urd` daemon and interact with it to define the required dataspaces, jobs, and I/O tasks. By relying on the interfaces in this API, it is possible for `urd` to have reliable information from trusted sources about what a process invoking the API should be allowed to do. This in turn allows it to: (1) account the usage that registered processes make of their assigned dataspaces; (2) reject any I/O task submissions from processes not registered in the service; and (3) reject any task submissions from registered processes involving dataspaces they shouldn't access. The `norns` user API, on the other hand, has been designed to allow parallel applications to query information about their configured dataspaces, and also to allow them to define, submit, monitor, and control I/O tasks. Listing 2 shows how a task to transfer a process buffer can be scheduled for asynchronous execution.

```c
void buffer_offloading(void* buffer, int size) {

  // define and submit transfer task for buffer
  norns_iotask_t tsk = NORNS_IOTASK(
        NORNS_IOTASK_COPY,
        NORNS_MEMORY_REGION(buffer, size),
        NORNS_POSIX_PATH(
            "tmp0://", "path/to/output"));

  if(norns_submit(&tsk) != NORNS_SUCCESS) {
     fatal("task submission failed");
  }

  work_not_dependent_on_task();

  // wait for task to complete and check status
  norns_wait(&tsk, /* timeout = */ NULL);
  norns_stat_t stats;
  norns_error(&tsk, &stats);

  if(stats.st_status == NORNS_ETASKERROR) {
    fatal("task failed");
  }
}
```

Listing 2. Defining, submitting, and monitoring a user-level I/O task

V. EVALUATION

A. Methodology

Testbed: All experiments were conducted using the NEXTGenIO prototype, which is composed of 34 compute nodes [38]. Each node has a dual Intel® Xeon® Platinum 8260M CPU @ 2.40 GHz (i.e. 48 cores per node), 192 GiB of RAM and 3 TBytes of Intel® DCPMM™ memory. Compute nodes are interconnected through an Omni-Path™ fabric, and a Lustre server (6 OSTs) is reached using a 56 Gbps InfiniBand™ link.

B. NORNS Performance

As an infrastructure service for job schedulers and applications, NORNS must add as few overhead as possible to the normal operations of its intended clients, and it is also important for NORNS to transfer data efficiently, so that jobs are not delayed due to staging operations. In this section, we evaluate how NORNS performs when serving concurrent requests from multiple clients as well as its data transfer rates when moving data between compute nodes.

Request Rate: Figures 4 and 5 show, respectively, NORNS performance when serving requests from local processes and from other NORNS instances in remote compute nodes. For local requests, we create up to 32 concurrent processes that submit 50×10^3 consecutive requests to the local `urd` daemon

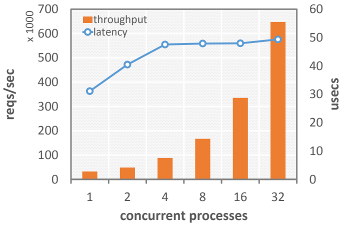


Fig. 4. NORNS throughput and latency serving local requests

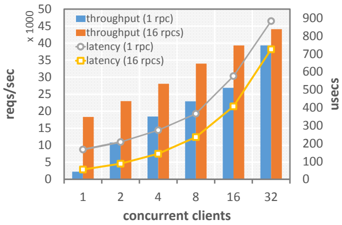


Fig. 5. NORNS throughput and latency serving remote requests

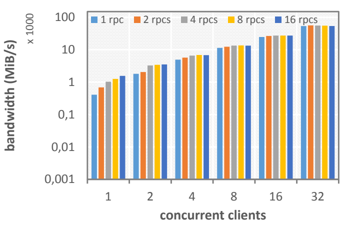


Fig. 6. NORNS aggregated bandwidth for remote data reads

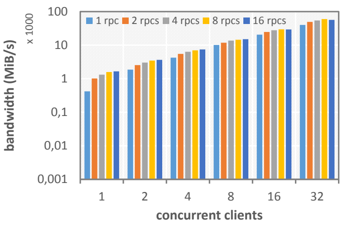


Fig. 7. NORNS aggregated bandwidth for remote data writes

using the norns API. For remote requests, we use up to 32 compute nodes to send 50×10^3 remote requests in parallel to the same NORNS target instance, both sequentially and in groups of 16. We configure NORNS to use the ofi+tcp plugin since it is less performant than other fabric-specific plugins and should be supported by most HPC clusters. In both cases we measure the latency taken by the daemon to respond to the client (i.e. the time taken to process the request, create a task descriptor, add it to the task queue, and respond to the client) as well as the observed aggregated throughput in requests per second (RPS). Each experiment is repeated 20 times. NORNS exhibits low latency when serving local requests (≈50 µseconds in the worst case), which is expectedly higher for remote requests (≈900 µseconds in the worst case) since they involve network communication rather than node-local IPCs. Throughput scales up to ≈700,000 local RPS and up to ≈45,000 remote RPS, which should be enough to support the expected load from Slurm and applications.

Transfer rate: Figures 6 and 7 show the data transfer rates when respectively reading/writing from/to remote NORNS instances. The benchmark measures the aggregated bandwidth rate from up to 32 clients reading/writing data in parallel from a single NORNS target. Again, we measure the performance using the ofi+tcp plugin and show results with 1 RPC and 16 RPCs in flight. NORNS clients use a 16 MiB buffer for transfers since increasing the buffer did not improve bandwidth further, and each experiment is repeated 20 times. Results show that aggregated bandwidth scales linearly with the number of nodes peaking at ≈55.6GiB/s for reads and at ≈59.7GiB/s for writes. A detailed examination of the results that the bandwidth per client saturates at ≈1.7GiB/s for reads and ≈1.8GiB/s for writes, and that it remains stable (without either rising or falling) even if the number of in-flight RPCs or nodes is increased. This suggests that NORNS Mercury-based *network manager* is capable of reaching the maximum bandwidth for this particular protocol and that transfer performance is limited by the underlying network stack.

C. Performance from Node-Local NVMs:

A key issue in supporting data-driven workflows effectively, is to exploit node-local I/O as much as possible to share data between workflow tasks, avoiding data transfers between storage layers and instructing the job scheduler to move computation to where data already resides. For this approach to be successful, the node-local devices used for absorbing normal application I/O need to provide a performance at least similar to that of the underlying PFS. In this experiment, we compare the performance between the Lustre PFS used in the NEXTGenIO cluster and the node-local NVMs available in compute nodes. We used IOR to use the 48 cores available to each node to spawn processes that created as many independent files, both using Lustre for storage and Intel®'s node-local DCPMM™s. The benchmark then proceeded to read/write from/to the created files sequentially, using a transfer size of 512KiB. To avoid benefiting from the page cache, file sizes were chosen to be larger than 192GiB to fill the nodes RAM. Both benchmarks were run for 25 independent repetitions during a maintenance period where fewer jobs competed for I/O resources. Figure 8 confirms, as expected, that the aggregated bandwidth from NVM devices is significantly higher than Lustre's median bandwidth, even up to an order of magnitude for higher node counts. It also scales better, even if Lustre is not under a production workload.

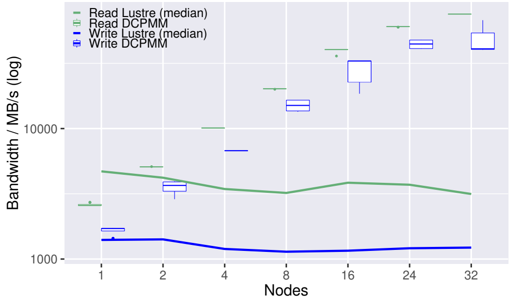


Fig. 8. Lustre I/O performance vs node-local Intel DCPMM NVM on the NEXTGenIO prototype

D. Workflow Performance

We also evaluated the performance benefits achievable when utilising the functionality we have outlined to maintain data on compute node NVMs and share that data between compute jobs in a workflow. We evaluated performance with a synthetic benchmark and using a user application (OpenFOAM), both of which will be discussed in the following sections.

Synthetic benchmark: We created a synthetic workflow benchmark that has a producer and a consumer of data, configurable to produce a range of files with a range of different sizes. We can run this benchmark either targeting the Lustre filesystem or the NVMs on each compute node and observe the performance difference between utilising both forms of storage. We can also utilise the scheduler integration

TABLE III
SYNTHETIC WORKFLOW BENCHMARK USING LUSTRE AND/OR NVMS IN A COMPUTE NODE

Component	Target	Runtime (seconds)
Producer	Lustre	96
Consumer	Lustre	74
Producer	NVM	64
Consumer	NVM	30

TABLE IV
SYNTHETIC WORKFLOW BENCHMARK WITH DATA STAGING

Component	Runtime (seconds)
Producer	64
Consumer	30
HPCG stage out	137
HPCG stage in	142
HPCG no activity	122

TABLE V
OPENFOAM WORKFLOW BENCHMARK USING LUSTRE VS NVMS + DATA STAGING

Workflow phase	Lustre	NVMs
decomposition	1191	1105
data-staging	–	32
solver	123	66

and NORNS components to maintain data on a compute node, in NVMs between workflow component runs and observe the performance benefit this provides. Table III outlines the performance achieved when producing and consuming 100GB of data running the workflow on Lustre or directly on NVMs. Each benchmark workflow ran 5 times and report the mean result. Performance varied by $<5\%$ across runs. Benchmarks were compiled using the Intel 19 compiler with the `-O3` flag.

For the benchmark targeting Lustre we ran the producer and consumer on two separate compute nodes to ensure that data/I/O caching in the computing node operating did not affect the measured runtimes. For the workflow using NVMs we ran in two different configurations, one with the components re-using the same node and the data stored on the NVMs, and one using different compute nodes with data moved off the producer node after production and pre-staged on to the consumer node before the consumer is run.

For the staging benchmark we run another application on the nodes where the data staging was occurring (both post-producer and pre-consumer staging) to evaluate the potential impact on applications of operations NORNS and the job scheduler may undertake. For this we chose to run the High Performance Conjugate Gradients (HPCG) [39] benchmark. This aims to represent the computational and data access patterns of a broad set of important computational simulation applications. The conjugate gradients algorithm used in the benchmark is not just floating point performance limited, it is also heavily reliant on the performance of the memory system, and to a lesser extend on the network used to connect the processors together. We ran a small HPCG test case that would complete in $\approx$122 seconds using 48 MPI processes per node.

We can see from the Table III, which outlines the performance of using Lustre directly, or running the workflow components consecutively on the same node, that using local NVM storage gives $\approx$46% faster performance (96 vs 170 seconds) overall runtime for the workflow compared to using Lustre. Note, for the NVM case we run a job that reads and writes 200GB of data between workflow components on the same node to ensure caching does not affect performance.

When considering the staging of data benchmark, were data is moved to Lustre after the producer has completed and loaded in to NVM before the consumer starts, we see outlined in Table IV that the Producer and Consumer tasks are not affected by this mode of operation, we achieve performance commensurate with running both tasks on the same node using node-local NVM. However, it is evident that the application running on the compute nodes whilst the data is being moved to and from the node-local storage is impacted by that activity. We experience an approximately 15% increase in runtime for the HPCG benchmark. It is worth noting that this is likely to be a disproportionately large impact compared to general applications as the runtime of HPCG is similar to that of the time required to stage the data to or from the NVM. For most applications this activity would normally only impact at the beginning and/or end of execution, when the job scheduler is preparing for a job finishing and the next job starting, and thus should impact a much smaller part of the application execution. However, it is important to note there is potential for impact on running applications.

OpenFOAM performance: OpenFOAM [40] is a C++ object oriented library for Computational Continuum Mechanics developed to provide Computational Fluid Dynamics functionality that can easily be extended and modify by users. It is parallelised with MPI and is heavily used in academia and industry for large scale computational simulations.

OpenFOAM often requires multiple stages to complete a simulation, from preparing meshes and decomposing them for the required number of parallel processes, to running the solver and processing results. It also, often, undertakes large amounts of I/O, reading in input data and producing data for analysis. It is common that the different stages require differing amounts of compute resources, with some stages only able to utilise one node, and others (such as the solver) requiring a large number of nodes to complete in a reasonable amount of time.

In general, OpenFOAM favours creating a directory per process that will be used for the solver calculations, necessitating a large amount of I/O for big simulation. Given these features, OpenFOAM is a strong target for both workflow functionality and improved I/O performance through node-local I/O hardware. For this benchmark we ran a low-Reynolds number laminar-turbulent transition modeling simulation of the flow over the surface of an aircraft [41], using a mesh with $\approx$43 million mesh points. We decomposed the mesh over 16 nodes enabling 768 MPI processes to be used for the solver step (picoFOAM). The decomposition step is serial, takes 1105 seconds, and requires 30GB of memory.

We ran the solver for 20 timesteps, and compared running the full workflow (decomposing the mesh and then running the solver) entirely using the Lustre filesystem or using node-local NVM with data staging between the mesh decomposition step and the solver. The solver produces 160GB of output data when run in this configuration, with a directory per process.

Running the solver using Lustre required 123 seconds, whereas running the solver using node-local storage required 66 seconds, close to two times faster (see Table V). Using node-local storage needs a redistribution of data from the storage on the single compute node used for decomposing the mesh to the 16 nodes needed for the solver. This data copy took 32 seconds, so even if not overlapped with other running tasks this approach would provide improved performance compared to directly using Lustre, more so when run for a full simulation, which would require many thousands of timesteps meaning the initial cost of copying the data would be negligible.

VI. Related Work

In [42], the authors discuss the potential benefit of using NVM in compute nodes and for computational science or data analytics workflows. The work presented in [43] also addresses the topic of HPC workflows and introduces NVStream, a user-level data management system for producer-consumer applications that exploits the byte-addressable and persistent nature of NVM to enable streamed I/O for scientific workflows. However, NVStream requires application modification and potentially algorithmic change to exploit the developed framework. Our approach provides the potential for applications to move towards new modes of I/O, i.e. through memory operations and sharing via NORNS, but also supports data movement through files for already existing applications.

ADIOS [44] provides functionality to undertake I/O across different storage targets, namespaces, and representations, including volatile memory. With recent extensions, including DataSpace/data staging integration, it can also manage and move data around or between systems. However, as with the NVStream approach, it is built around object or key-value level storage where the middleware stores data for the application. It does not provide the same functionality to support and move files that NORNS supports, and also requires application changes to exploit the provided functionality. Furthermore, the data staging functionality is designed in a *many-to-few* operation mode, with compute node data written to stage nodes. This means the overall buffer available for data staging is limited, and subject to performance interference between applications. Our approach leverages storage within compute nodes to reduce application interference and scale I/O performance. It can also be used in scenarios where data is staged to a limited number of nodes or to an external resource such as a burst buffer, but is not restricted to this mode of operation. Kimpe et al. [45] build atop IOFSL [46] to offer a container abstraction and API to help manage local storage devices. Similarly to NORNS, it focuses on data movement across the system but with the goal of optimizing checkpointing I/O instead of workflow-related I/O. Swift [47][48] is a workflow description language and manager that allows describing data flows in terms of composable independent computational activities. Swift manages each workflow independently and is tailored for scientific campaigns involving multi-site transfers, whereas NORNS interfaces with SLURM to provide optimal data transfers within the HPC cluster.

Work has also been done into integrating burst buffer devices into scheduling and resource management decisions. For example Cray DataWarp users can request allocations in their job request through the DataWarp API [9], and IBM burst buffers allow staging data into and out of node-local storage via their job script [49]. Some work involving data-aware scheduling has been done for grid computing [50], and some more recent works have shown ways to map jobs onto compute nodes [51] and to integrate storage with batch job schedulers [52][53][54], but neither explicitly look at optimizing end-to-end scientific workflows. Similarly to NORNS, CLARISSE [55] provides middleware functionality aiming at optimizing I/O operations on a system-wide basis. CLARISSE does not consider node-local storage resources or full workflow/workload scheduling optimizations, however. TRIO [19] explores how to efficiently move large checkpointing datasets to the PFS by utilizing the burst buffers. Data Elevator [20] and Stacker [21] are similar to NORNS in that they focus on asynchronously moving data across I/O layers to optimize scientific workflows. The former specializes on applications using HDF5 while the latter optimizes data movements using machine learning techniques. As far as we know, neither integrates with the scheduler to capture a global view of the workflows in the system.

VII. Conclusions and Future Work

We present the design and implementation of NORNS, a new service for asynchronous data staging that coordinates with job schedulers to orchestrate data transfers to achieve a better utilization of the multiple HPC I/O layers. We also introduce several extensions to Slurm that allow users to express data dependencies and I/O requirements to their job definitions which, coupled with the APIs and primitives provided by NORNS, effectively adds support to Slurm for modelling and executing data-driven workflows. Experimental results show that by leveraging this framework to include awareness of node-local NVMs storage into Slurm's scheduling process, makes better use of the available storage layers, reducing contention to the PFS by effectively isolating workflow I/O to node-local storage and reducing job runtimes.

Nonetheless, we acknowledge that there is still the challenge of how to optimally schedule transfers from/to the PFS/between compute nodes to maximize job throughput and minimize I/O interference with user applications. NORNS instances in each compute node could monitor the performance of the dataspaces assigned to them, and arbitrate pending requests to minimize metrics such as time to completion, energy consumption, or I/O task throughput. Information about observed I/O performance could be fed back to the job scheduler so that it could take better informed decisions, offering a robust platform upon which smarter data-aware scheduling algorithms could be built. Integrating NORNS with existing workflow engines, exploring the effects of competition between administrative and application I/O as well as implementing transfer plugins for shared burst buffers are additional tasks for the future.

REFERENCES

[1] "TOP500 Supercomputer Sites," November 2018. [Online]. Available: https://www.top500.org/lists/2018/11/

[2] Lister, JB and Duval, BP and Farthing, JW and Fredian, TJ and Greenwald, M and How, J and Llobet, X and Saint-Laurent, F and Spears, W and Stillerman, JA, "The ITER project and its data handling requirements," in *9th ICALEPCS Conference, Gyeongju, Korea*, 2003.

[3] "The Large Hadron Collider." [Online]. Available: http://home.cern/topics/large-hadron-collider

[4] CMS Collaboration *et al.*, "The CMS experiment at the CERN LHC," 2008. [Online]. Available: http://cms.cern/

[5] P. Carns, K. Harms, W. Allcock, C. Bacon, S. Lang, R. Latham, and R. Ross, "Understanding and improving computational science storage access through continuous characterization," *ACM Transactions on Storage (TOS)*, vol. 7, no. 3, p. 8, 2011.

[6] S. Habib, R. Roser, R. Gerber, K. Antypas, K. Riley, T. Williams, J. Wells, T. Straatsma, A. Almgren, J. Amundson *et al.*, "ASCR/HEP exascale requirements review report," *arXiv preprint arXiv:1603.09303*, 2016.

[7] P. A. Abell, D. L. Burke, M. Hamuy, M. Nordby, T. S. Axelrod, D. Monet, B. Vrsnak, P. Thorman, D. Ballantyne, J. D. Simon *et al.*, "LSST Science Book, Version 2.0," *arXiv:0912.0201*, 2009.

[8] E. Deelman, T. Peterka, I. Altintas, C. D. Carothers, K. K. van Dam, K. Moreland, M. Parashar, L. Ramakrishnan, M. Taufer, and J. Vetter, "The future of scientific workflows," *The International Journal of High Performance Computing Applications*, vol. 32, no. 1, pp. 159–175, 2018.

[9] CRAY Inc., "libdatawarp - the DataWarp API." [Online]. Available: https://pubs.cray.com/content/S-2558/CLE%206.0.UP06/xctm-series-datawarptm-user-guide/libdatawarp---the-datawarp-api

[10] DataDirect Networks (DDN®) Storage, "Infinite Memory Engine." [Online]. Available: https://www.ddn.com/products/ime-flash-native-data-cache/

[11] National Energy Research Scientific ComputingCenter (NERSC), "Cori Computational System." [Online]. Available: https://www.nersc.gov/users/computational-systems/cori/

[12] Los Alamos National Laboratory (LANL), "The Trinity supercomputer." [Online]. Available: https://www.lanl.gov/projects/trinity/

[13] Intel Corporation, "Optane™ memory." [Online]. Available: https://www.intel.com/content/www/us/en/architecture-and-technology/optane-memory.html

[14] J. L. Whitt, "Oak Ridge Leadership Computing Facility: Summit and Beyond," 3 2017. [Online]. Available: https://indico.cern.ch/event/618513/contributions/2527318/attachments/1437236/2210560/SummitProjectOverview_jlw.pdf

[15] Lawrence Livermoore National Lab, "Sierra." [Online]. Available: https://hpc.llnl.gov/hardware/platforms/sierra

[16] Barcelona Supercomputing Center, "MareNostrum IV – Technical Information." [Online]. Available: https://www.bsc.es/marenostrum/marenostrum/technical-information

[17] A. M. Caulfield, L. M. Grupp, and S. Swanson, "Gordon: using flash memory to build fast, power-efficient clusters for data-intensive applications," *ACM Sigplan Notices*, vol. 44, no. 3, pp. 217–228, 2009.

[18] "Slurm Burst Buffer Guide." [Online]. Available: https://slurm.schedmd.com/burst_buffer.html

[19] T. Wang, S. Oral, M. Pritchard, B. Wang, and W. Yu, "Trio: Burst Buffer Based I/O Orchestration," in *2015 IEEE International Conference on Cluster Computing*. IEEE, 2015, pp. 194–203.

[20] B. Dong, S. Byna, K. Wu, H. Johansen, J. N. Johnson, N. Keen *et al.*, "Data elevator: Low-contention Data Movement in Hierarchical Storage Systems," in *2016 IEEE 23rd International Conference on High Performance Computing (HiPC)*. IEEE, 2016, pp. 152–161.

[21] P. Subedi, P. Davis, S. Duan, S. Klasky, H. Kolla, and M. Parashar, "Stacker: An Autonomic Data Movement Engine for Extreme-scale Data Staging-based in-situ Workflows," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*. IEEE Press, 2018, p. 73.

[22] A. B. Yoo, M. A. Jette, and M. Grondona, "Slurm: Simple linux utility for resource management," in *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer, 2003, pp. 44–60.

[23] A. Shoshani and D. Rotem, *Scientific data management: challenges, technology, and deployment*. Chapman and Hall/CRC, 2009.

[24] B. Dong, X. Li, L. Xiao, and L. Ruan, "A new file-specific stripe size selection method for highly concurrent data access," in *Proceedings of the 2012 ACM/IEEE 13th International Conference on Grid Computing*. IEEE Computer Society, 2012, pp. 22–30.

[25] P. Braam, "The Lustre Storage Architecture," Cluster File Systems Inc. architecture, design, and manual for Lustre, 11 2002. [Online]. Available: http://www.lustre.org/docs/lustre.pdf

[26] F. B. Schmuck and R. L. Haskin, "GPFS: A Shared-Disk File System for Large Computing Clusters." in *FAST*, vol. 2, no. 19, 2002.

[27] J. Shalf, S. Dosanjh, and J. Morrison, "Exascale computing technology challenges," in *International Conference on High Performance Computing for Computational Science*. Springer, 2010, pp. 1–25.

[28] EPCC, The University of Edinburgh, "ARCHER national supercomputing service." [Online]. Available: https://www.epcc.ed.ac.uk/facilities/archer

[29] "HPC IO Benchmark Repository." [Online]. Available: https://github.com/hpc/ior

[30] J. Lofstead, F. Zheng, Q. Liu, S. Klasky, R. Oldfield, T. Kordenbrock, K. Schwan, and M. Wolf, "Managing variability in the IO performance of petascale storage systems," in *SC'10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, 2010, pp. 1–12.

[31] B. Xie, J. Chase, D. Dillow, O. Drokin, S. Klasky, S. Oral, and N. Podhorszki, "Characterizing output bottlenecks in a supercomputer," in *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2012, pp. 1–11.

[32] A. Kougkas, H. Devarajan, X.-H. Sun, and J. Lofstead, "Harmonia: An Interference-Aware Dynamic I/O Scheduler for Shared Non-Volatile Burst Buffers," in *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, 2018, pp. 290–301.

[33] Y. Hashimoto and K. Aida, "Evaluation of performance degradation in hpc applications with vm consolidation," in *2012 Third International Conference on Networking and Computing*. IEEE, 2012, pp. 273–277.

[34] J. Lofstead and R. Ross, "Insights for exascale io apis from building a petascale io api," in *SC'13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE, 2013, pp. 1–12.

[35] D. A. Reed and J. Dongarra, "Exascale computing and big data," *Communications of the ACM*, vol. 58, no. 7, pp. 56–68, 2015.

[36] "Protocol Buffers." [Online]. Available: https://developers.google.com/protocol-buffers/

[37] J. Soumagne, D. Kimpe, J. A. Zounmevo, M. Chaarawi, Q. Koziol, A. Afsahi, and R. B. Ross, "Mercury: Enabling remote procedure call for high-performance computing," in *2013 IEEE International Conference on Cluster Computing (CLUSTER), Indianapolis, IN, USA, September 23-27, 2013*, 2013, pp. 1–8.

[38] EPCC (The University of Edinburgh), Intel, Fujitsu, Barcelona Supercomputing Center, Technische Universität Dresden, Allinea, ECMWF, ARCTUR, "Next Generation I/O for the Exascale." [Online]. Available: https://www.nextgenio.eu

[39] J. J. Dongarra, M. A. Heroux, and P. Luszczek, "Hpcg benchmark : a new metric for ranking high performance computing systems," Knoxville, Tennessee, Tech. Rep. UT-EECS-15-736, November 2015.

[40] H. Jasak, "Openfoam: Open source cfd in research and industry," *International Journal of Naval Architecture and Ocean Engineering*, vol. 1, no. 2, pp. 89 – 94, 2009. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S2092678216303879

[41] Andrejašič, Matej and Veble, Gregor and Bat, Nejc, "Cloud-based Simulation of Aerodynamics of Light Aircraft." [Online]. Available: https://hpc-forge.cineca.it/files/CoursesDev/public/2015/Workshop_HPC_Methods_for_Engineering/cloud_based_aircraft.pdf

[42] M. Weiland, A. Jackson, N. Johnson, and M. Parsons, "Exploiting the performance benefits of storage class memory for hpc and hpda workflows," *Supercomputing Frontiers and Innovations*, vol. 5, no. 1, 2018. [Online]. Available: http://superfri.org/superfri/article/view/164

[43] P. Fernando, A. Gavrilovska, S. Kannan, and G. Eisenhauer, "Nvstream: Accelerating hpc workflows with nvram-based transport for streaming objects," in *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '18. New York, NY, USA: ACM, 2018, pp. 231–242. [Online]. Available: http://doi.acm.org/10.1145/3208040.3208061

[44] J. F. Lofstead, S. Klasky, K. Schwan, N. Podhorszki, and C. Jin, "Flexible io and integration for scientific codes through the adaptable io system (adios)," in *Proceedings of the 6th International Workshop*

on Challenges of Large Applications in Distributed Environments, ser. CLADE '08. New York, NY, USA: ACM, 2008, pp. 15–24. [Online]. Available: http://doi.acm.org/10.1145/1383529.1383533

[45] D. Kimpe, K. Mohror, A. Moody, B. Van Essen, M. Gokhale, R. Ross, and B. R. de Supinski, "Integrated in-system storage architecture for high performance computing," in *Proceedings of the 2nd International Workshop on Runtime and Operating Systems for Supercomputers*. ACM, 2012, p. 4.

[46] N. Ali, P. Carns, K. Iskra, D. Kimpe, S. Lang, R. Latham, R. Ross, L. Ward, and P. Sadayappan, "Scalable I/O forwarding framework for high-performance computing systems," in *2009 IEEE International Conference on Cluster Computing and Workshops*. IEEE, 2009, pp. 1–10.

[47] Y. Zhao, M. Hategan, B. Clifford, I. Foster, G. Von Laszewski, V. Nefedova, I. Raicu, T. Stef-Praun, and M. Wilde, "Swift: Fast, reliable, loosely coupled parallel computation," in *2007 IEEE Congress on Services (Services 2007)*. IEEE, 2007, pp. 199–206.

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

[49] IBM, "IBM/CAST - Cluster Administration and Storage Tools." [Online]. Available: https://github.com/IBM/CAST

[50] J. M. Schopf, "A general architecture for scheduling on the grid," *Special issue of JPDC on Grid Computing*, vol. 4, 2002.

[51] M. Mubarak, C. D. Carothers, R. B. Ross, and P. Carns, "Enabling parallel simulation of large-scale hpc network systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 1, pp. 87–100, 2016.

[52] J. Bent, D. Thain, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and M. Livny, "Explicit control in the batch-aware distributed file system." in *NSDI*, vol. 4, 2004, pp. 365–378.

[53] A. Gainaru, G. Aupy, A. Benoit, F. Cappello, Y. Robert, and M. Snir, "Scheduling the i/o of hpc applications under congestion," in *2015 IEEE International Parallel and Distributed Processing Symposium*. IEEE, 2015, pp. 1013–1022.

[54] S. Herbein, D. H. Ahn, D. Lipari, T. R. Scogland, M. Stearman, M. Grondona, J. Garlick, B. Springmeyer, and M. Taufer, "Scalable i/o-aware job scheduling for burst buffer enabled hpc clusters," in *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*. ACM, 2016, pp. 69–80.

[55] F. Isaila, J. Carretero, and R. Ross, "Clarisse: A middleware for data-staging coordination and control on large-scale hpc platforms," in *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, May 2016, pp. 346–355.