Evaluating On-Demand Parallel File System Impacts on Compute-Bound Tasks

1st Matthew L. Curry
Center for Computing Research
Sandia National Laboratories
Albuquerque, New Mexico, USA
mlcurry@sandia.gov

2nd Michael Aguilar
High Performance Computing Systems
Sandia National Laboratories
Albuquerque, New Mexico, USA
mjaguil@sandia.gov

3rd Shyamali Mukherjee
Computation and Analysis for National Security
Sandia National Laboratories
Livermore, California, USA
smukher@sandia.gov

***Abstract*—Current HPC systems are limited in performance and architecture by the the stranding of disaggregated components. In current HPC systems, common tools do not exist to manage both network fabrics and component resources, such as, CPUs, hierarchical memory components, and accelerators. Due to lack of access to good resource management tools, current HPC systems suffer from inefficient over-design where resources must be provided locally, often in node, to make In instances that HPC running jobs could potentially require the resources. In addition, the inability to dynamically share available resources, as they are required by running batch jobs can lead to reduced and inefficient computational performance and running job failure. Centralized resource management can potentially mitigate, Out-of-Memory conditions, IO thashing, stranding of available resources, such as, CPUs, GPUs, and memories, and provide dynamic network fail-over. It is clear that resource management, using a standardized interface, can enable clients to monitor, compose, and intelligently provision resources, in beneficial ways.**

The OpenFabrics Management Framework (OFMF) is a open sourcec resource manager being developed by the OpenFabric Alliance, the DMTF, SNIA, and the CXL Consortium. The OFMF implements Redfish and Swordfish through the implementation of a Swordfish Endpoint Emulator and network Agents. The Emulator provides centralized resource monitoring and command control for the attached hardware resources and network fabrics, through matching network Agents. A Composability Manager, integrated with the OFMF, can mitigate stranded resources by providing a method for sharing hardware, CPUs, GPUs, and memories. Integration with the OFMF provides the capability of dynamic provisioning of resources to maintain running client computations.

***Index Terms*—component, formatting, style, styling, insert**

I. Introduction

Classically, parallel file systems have been a global resource shared among all tasks running on an HPC system. While there have been many efforts to reduce interference between jobs [?], [?], these methods have generally not been able to achieve high-quality performance isolation. While new storage technologies have made parallel file systems faster, users are often frustrated at performance instability stemming from storage, especially for data analytics workloads.

Over the past decades, this phenomenon has become widely understood, and most HPC users now have firm experiences guiding what performance isolation characteristics are for each resource in an HPC system. In Table I, we provide an overview of what a user expects from tasks fitting different performance profiles. In particular, all tasks that are allocated space-shared resources like compute nodes are expected to achieve good performance isolation from other tasks on the system, or even within the same job. However, globally shared resources like parallel file systems have no guarantees about performance stability. This has led to assumptions baked into application architectures. For instance, bulk synchronous parallel tasks have long assumed equal performance between compute nodes, with minimal delays when completing local computation steps. Such applications can be very efficient, but tend to be tightly coupled and intolerant of small delays during computation.

The advent of dynamic parallel file systems, like Lustre On Demand (LOD) [?] and BeeGFS On-Demand (BeeOND) [?], has created a new class of storage that can provide performance isolation between different allocations in a system. This is accomplished by aggregating storage devices (like onboard SSDs) distributed on many or all compute nodes in a job. Storage service daemons, for object storage, file system management, and metadata operations, are distributed on each node. This provides a user with a private file system free of interfering access from other users.

While new storage performance isolation expectations can now be met, previously held assumptions are now at risk of being violated. Storage server processess do not usually demand high CPU utilization during normal operations, but tightly coupled applications are prone to performance variation when subject to even small interruptions [?]. When on-demand file systems are spread across all nodes in a job, all nodes are used to provide storage services for any tasks performing file I/O, consuming resources previously assumed to be dedicated to unrelated tasks. This poses a concrete problem when mixing several tasks of different profiles within a single job, e.g. a data analytics task alongside a tightly coupled simulation. Data accesses from the analytics task will incur CPU, memory, and network use on hosts running the tightly coupled simulation, potentially impacting some processes more than others.

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

TABLE I
AN OVERVIEW OF SEVERAL PERFORMANCE PROFILES, EXAMPLE LEADING REPRESENTATIVE BENCHMARKS, AND DEGREE OF PERFORMANCE ISOLATION TYPICALLY EXPECTED BETWEEN NODES AND/OR TASKS BY HPC USERS.

Profile	**Description**	**Benchmark**	**Isolation**
CPU-bound	Heavy use of CPU and accelerators	HPL [?]	Strong
Memory-bound	Reads and writes to main memory	STREAM [?], HPCG [?]	Strong
Network-bound	Sending and receiving data among nodes in a task	Intel MPI Benchmarks [?]	Medium-to-Strong
IOPs-bound	Many small reads/writes to a few files	IOR-hard [?]	Weak
Bandwidth-bound	Large reads/writes to a few files	IOR-easy [?]	Weak
Metadata-bound	Many small reads/writes to may files	mdtest [?]	Weak

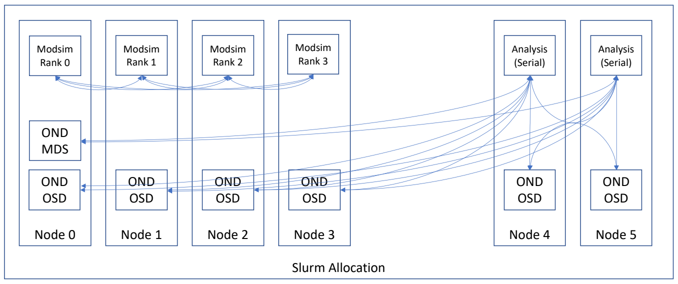


Fig. 1. Otherwise unrelated tasks using an on-demand file system can cause traffic and associated work on nodes running other tasks.

Figure 1 illustrates this isolation issue. Three different jobs are colocated in a single Slurm allocation: one tightly coupled modeling and simulation task, and two non-communicated data analysis tasks. There can be many reasons why these jobs are colocated, including the case that the data analysis is being performed on output from the simulation's checkpoint-restart phase. During the simulation's compute-heavy phase, there will be nearly constant traffic between the analysis jobs and the OSDs. An even more impactful interaction is the potential n-to-1 communcation with MDS processes. If the analytics jobs have occasion to do work following the metadata-bound performance profile, the metadata service can be made quite busy.

In this paper, we will discuss our experiments with compute-bound and storage-bound benchmarks to quantify possible compute interference generated by on-demand file system use. These experiments target BeeOND running on a 288-node HPC platform with on-board SSDs. Highly regular benchmarks like HPL and IOR are composed as tasks in a single job to orchestrate worst-case interference, allowing us to start evaluating performance trade-offs. We will also discuss interference mitigation techniques.

II. RELATED WORK

As HPC systems have grown, their associated storage systems have been widely identified as a potential bottleneck. One major concern was the viability of the checkpoint-restart pattern of resilience, as compute and memory progress was outpacing parallel file system performance. However, the increasing prominence of data analytics workloads on HPC system presented further risk to the viability of upcoming system architectures, yielding an increased emphasis on the search for alternative solutions.

One early solution was to create technologies specifically to increase the performance of checkpointing, even at the expense of later restarts. Solutions in this space included PLFS [?], Zest [?], and buddy-checkpointing via SCR [?]. These techniques were often regarded as stop-gap solutions, as they decreased checkpoint reliance (or checkpoint requirements) on parallel file systems. However, these techniques did not address the core difficulty of parallel file systems struggling to meet new performance requirements.

The next step was development of new types of storage systems to allow storage workloads to largely avoid parallel file system use. A new class of technology termed "burst buffers" was developed [?]. Burst buffers are typically flash-based storage appliances distributed throughout an HPC system, designed to service workloads in a more local fashion. Space within them are often allocated via the job scheduler, allowing some level of performance isolation from other jobs using other burst buffer components in remote parts of the system. Although sometimes still called burst buffers, this term has fallen out of favor because they are now more widely used than the original application, absorbing bursts of I/O generated by checkpoint restart. Instead, these systems (like DataWarp and DDN Infinite Memory Engine) are also used extensively for data analytics, helping resolve the original shared parallel file system contention problem.

A relatively new development is the on-demand parallel file system. BeeGFS-On-Demand [?] is a commonly available instance of this technology. Lustre On Demand [?] is a forthcoming design based on this concept. The main idea is to assemble node-local resources (like on-node SSDs and NVMe) to create a parallel file system independent of the centralized file system. This is accomplished by launching file system daemons within each node to serve requests from the clients. One common deployment allows for individual HPC jobs to assemble their own parallel file systems, eliminating parallel file system contention entirely.

While managing parallel file system contention has been a widely discussed topic [?], the impact of I/O processes on compute-bound tasks has not yet been deeply explored. Microkernel research from past decades has highlighted the impact of daemons commonly found in Linux on very large scale, tightly coupled jobs [?].

III. Methods

A. Implementation Decisions Made on our BeeOND Usage

We have started design work to provide HPC users with their own private BeeOND filesystems that are isolated from a common shared parallel filesystem. For production HPC use, we wanted to implement composable and ephemeral filesystems that were made up of Workload Manager allocated compute node storage. We made the decision that we wanted the data on the filesystems to last only as long as an individual user had use of a set of compute nodes. Once a user job allocation was completed, the storage would be cleared out for new job allocations using the same nodes. The advantage of clearing the node-local storage was that it better secured data that should not be shared and it would allow the nodes to quickly attach to new storage configurations. Further, we decided that we wanted to implement a parallel file storage solution where every allocated compute node would both contribute to the storage and be a client to a private storage solution. Thus, we designed our BeeOND filesystem around a node-local parallel filesystem architecture. Another aspect of our design was the fact that we wanted our BeeOND filesystem to attach to RDMA for lower latency and better file transfer performance. Finally, we wanted our BeeOND filesystem to be available to users running both batch jobs and interactive jobs.

Our on-site production HPC systems use the well-supported and common Slurm Workload Manager for a compute node allocation scheduler. Integration with the Slurm Workload Manager provided our BeeOND filesystem design with several convenient contributions for composing and managing private BeeOND filesystems. For instance, Slurm Prolog and Epilog scripts are designed to run in parallel. We wanted the start-up and tear-down of the BeeOND filesystem to be insignifcant regardless of the scale of node allocation quantity. By careful use of Slurm Prolog and Epilog, we were able to create parallel component instances that were assembled into complete stable private BeeOND filesystems in under 3 seconds and disassembled and erased in under 6 seconds, regardless of the scale of the compute node allocation.

We wanted creation of private user BeeOND filesystems to become an automatic part of the Slurm Workload Manager allocation. We found that the normal implementation of BeeOND would not fit our needs. Under the normal BeeOND filesystem operations, a user would need to manually start the BeeOND filesystem. In our production implementation we deemed that process to be unacceptable for ease of use and convenience. However, close integration of the BeeOND components into a prescribed set of serialized start-up events would make creation and deletion of the filesystems an automatic feature of Slurm allocation and deallocation.

Integration with Slurm provided us with another positive feature for private filesystem management. Slurm contains error and fault-tolerance infrastructure so that we would be able to handle unexpected filesystem issues. Slurm gracefully provides error handling and logging. In the event that the private filesystem failed to be properly started due to a hardware-related issue, the Slurm Workload Manager would be notified, logs detailing the issue would be generated, and the compute nodes would be drained for further inspection.

B. Integration of the BeeOND filesystem with Slurm

```
images/cluster_layout.png
```

Fig. 2. Planned Burst-Buffer Architecture

Our production private BeeOND filesystem architecture was completed as shown in figure 2. Slurm aided us in providing data locality to allocated job runs with Slurm partition information on our filesystem topology and the fact that Slurm has a built-in affinity for contiguous node allocation.

Slurm provides useful run-time variable information that is passed through the Slurm function *slurmstepd*. In our implementation, we wanted to provide HPC users with the ability to toggle on-and-off usage of the BeeOND filesystem. This was designed into our Slurm Prolog script with a Slurm variable using the Slurm 'Constraint' feature. In our Slurm Prolog script, the batch variable *SLURM_JOB_CONSTRAINTS* was checked for the value *beeond*. If the constraint *beeond* value was set, a log message was passed to standard node logging to reflect that a private BeeOND filesystem was user requested and was being used.

We wanted each node in the BeeOND filesystem to know the role it had in the collective parallel filesystem. Notification of the individual compute nodes was accomplished with the Slurm allocation variable *SLURM_NODELIST*. Passing *SLURM_NODELIST* inside Slurm Prolog with *hostlist* allowed our Prolog script parser to determine the individual filesystem roles of the compute node in the node-local BeeOND filesystem. In our implementation, the *SLURM_NODELIST* variable was copied to an internal variable in each parallel

Slurm Prolog script. Once the internal variable was loaded with the variable value of Slurm allocated nodes, *hostlist* was used to deconstuct the variable into a parsable compute node list. Next, the hostname of the compute node was checked against the compute node list. Using pre-determined architectural decisions, roles were assigned to each compute node in the Slurm allocation and services were started to create the BeeOND filsystem.

C. Customized BeeOND Filesystem Management with Parameter Passing

As part of the design of our BeeOND filesystem, we separated out the BeeOND filesystem management into separate start and stop scripts that were called by the Slurm Prolog and Epilog scripts. The advantage of separate management scripts was that we found that we could create more versatility in our filesystem architecture in future deployments on other HPC systems. Also, we like the fact that we would have modularity in our start and stop functions to help us with software debugging.

We wanted our BeeOND filesystem storage to be optimized for many different HPC architectures and different quantities of metadata and storage servers. Through careful analysis of the pre-packaged BeeOND shell scripts *beeond start* and the *beeond stop* scripts, we discovered that the key pieces of the pre-packaged BeeOND start-up script consisted of initialization and stoppage of just a few filesystem component services. With the knowledge that we gained from our script study, we were able to write our own custom BeeOND filesystem management software that fulfilled our needs for automatic filesystem composition and decomposition. Further, our custom start and stop scripts provided us complete versatility in quantity of MDT and OST services and node placement.

With our new custom scripts, in our BeeOND implementation, a Mangement server (Mgmt) component was the first service to run. Mgmt calls were completed with a *Mgmtd* storage directory, log file info, PID file, connection port, and as a daemon. We chose the lowest node in each *SLURM_NODELIST* to be the Mgmtd server.

The second BeeOND filesystem component in our BeeOND filesystem was the storage server. Every BeeOND filesystem consists of even striping of data across a user-defined quantity of OSTs. For our first BeeOND implementattion, we decided to make every node in our allocation a single OST on a local OSS server. Each storage server call requires a Storage Store directory, log file information, a PID file, the Mgmtd connection port, the name of the Mgmtd server, and as a daemon.

Again, the lowest entry in the *SLURM_NODELIST* became the default Metadata sever for our usage. We did implement inherent capabilities into our custom scripting to allow us to alter the number of Metadata servers and placement of the servers, if we would like to modify those options, in the future. Provided in our Metadata server start-ups are the name of the Mgmtd server, a Meta Store directory, log file information, a connection port, and daemonization.

Each node was a client of the node-local BeeOND parallel filesystem. BeeOND implements the mount with the use of a *helperd* service. Provided to *helperd* was the name of the Mgmtd server and the connection port. Once the *helperd* service was started, a *beeond_mount* command was called and the private BeeOND filesystem was mounted to /mnt/beeond.

The final architectural layout of the node-local BeeOND filesystem is shown in figure 3. The lowest node in the allocation became the Mgmt server, the Metadata server, an OST, and a client. The other nodes in the Slurm allocation became both OST servers and clients.

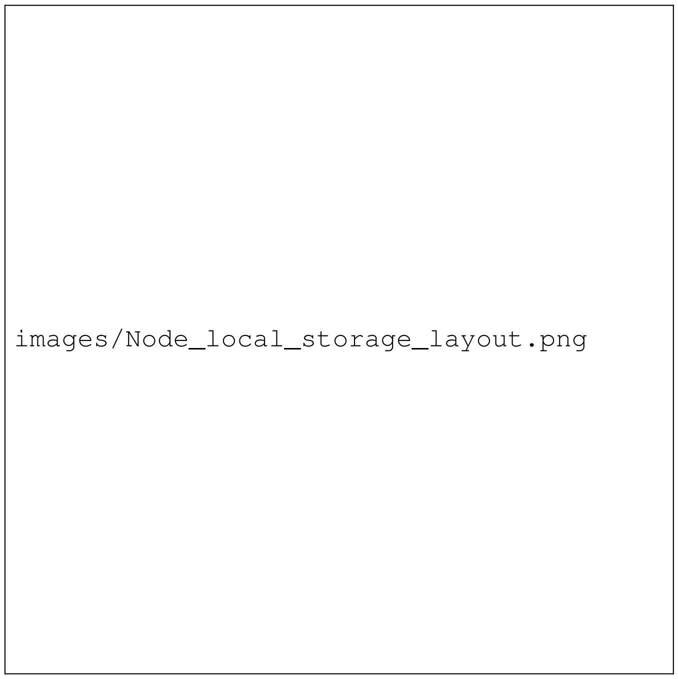


Fig. 3. Node Local Burst Buffer Architecture

Once an HPC allocation was completed, the BeeOND filesystem process components were matched to each node-local storage mount using the same methods that were used in the Slurm Prolog scripts. After node identification, as part of the Slurm Epilog script, parameters were passed to an individual BeeOND stop script on every node in the private BeeOND filesystem. A kill signal was sent to each compute node with a fuser command. A polling check insured that the processes were stopped before an XFS reformat to the node-local file storage. After any BeeOND processes were stopped, the back-end XFS storage was reformatted and remounted for use in another potential Slurm allocation.

D. Production HPC System Installation

Our first implementation of the new node-local private BeeOND filesystem design was on an ARM64 HPC system. The system consisted of a dual socket ThunderX2 processor with Socket Direct Nvidia/Mellanox 100 Gb/s EDR InfiniBand HCA ports and switches. Each node contained a 1 TB SATA interface SSD (HPE Part Number: 875852-001).

We created a single 894GB partition on each SSD drive and formatted the drive as an XFS block device. The BeeOND filesystem requires that the underlying block device support extended attributes. XFS filesystem provides extended attributes and is the standard filesystem for Red Hat 7.x and Red Hat 8.x Operating Systems. As part of preparing each SSD block device for BeeOND use, we wrote a special UDEV rule that examined the partitions on the SSD device to make sure that only a single continuous partition of 894GB was on the device. Next, our UDEV rule would assign the expected SSD, on */dev/sda1* to read/write permissions and create a symbolic device */dev/beeond_store* to denote success of the readiness of the device for our node-local BeeOND filesystem block storage. In the event of a failure of the UDEV rule, the compute node would not be put into Slurm queue and would be listed as not available.

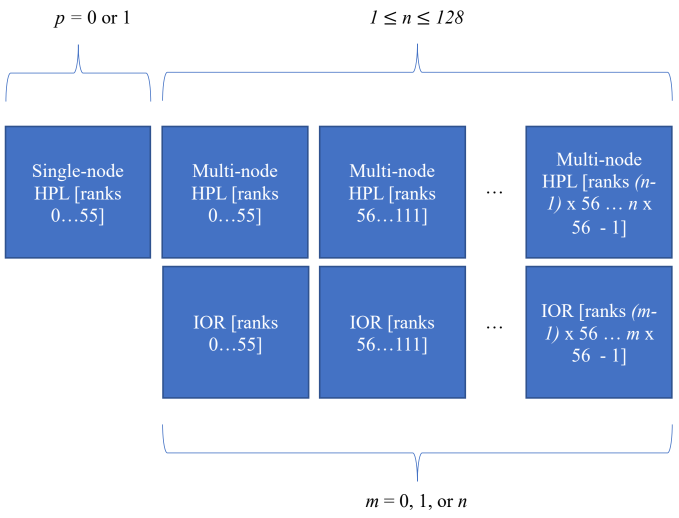
Our Production HPC systems use custom *nodeup* and *node-down* scripts to examine the compute nodes for hardware and software issues, prepare the compute nodes for Slurm allocations, and prepare the compute node shared storage mounts. We added our BeeOND node-local compute node mounts and the *beeond* device driver management to the custom scripts. In the event of a device driver mismatch to the kernel, BeeOND can do run-time re-compiling of the kernel module. We incorporated the recompiling feature into our node initialization script. Finally, the scripts mounted */dev/beeond_store* on mount */beeond* readying the SSD for our node-local filesystem.

E. *Experimental Procedure*

For this study, an experiment is a multi-node HPL task run in the same compute allocation with an IOR task of various sizes. These tasks are placed on non-overlapping sets of nodes, creating a situation without any explicit CPU contention. By measuring the HPL completion time of each configuration, we will be able to detect significant runtime impacts.

Figure 4 demonstrates how tasks are laid out in an allocation. From this basic structure, we created a set of five experiment classes to measure different phenomenon.

- **HPL-Only:** $k = 0, m = 0$. This is the control experiment that excludes IOR as a factor. Since this shared experimental infrastructure, BeeOND daemons were configured and started, but the only task running in the compute allocation is an n-node HPL job.
- **Matching Lustre:** $k = 0, m = n$. This can be considered another type of control experiment that excludes BeeOND as a factor, while still allowing IOR to potentially demonstrate an impact. This is valuable to determine whether the file system traffic from IOR can perturb an HPL running on other allocated nodes. Crucially, this is the only configuration that does not load any BeeOND daemons.
- **Single BeeOND:** $k = 0, m = 1$. This demonstrates the impact of including a single node running a data-intensive workload.
- **Matching BeeOND:** $k = 0, m = n$. This demonstrates the impact of a larger number of data-intensive processes on HPL performance.
- **Matching BeeOND (no meta):** $k = 1, m = n$. This demonstrates the same type of potential impact as the Matching BeeOND test, but explicitly places a separate task on the same node as the metadata server. This allows us to ensure that the multi-node HPL is not colocated with the metadata server or other management servers we place alongside it. Therefore, the multi-node HPL will only experience performance impacts from the object storage server.



p = 0 or 1
$1 \leq n \leq 128$
Single-node HPL [ranks 0…55]
Multi-node HPL [ranks 0…55]
Multi-node HPL [ranks 56…111]
…
Multi-node HPL [ranks (n-1) x 56 … n x 56 - 1]
IOR [ranks 0…55]
IOR [ranks 56…111]
…
IOR [ranks (m-1) x 56 … m x 56 - 1]
m = 0, 1, or n

Fig. 4. An illustration demonstrating possible configurations of the experiment.

1) HPL: We used HPL [?] as a well-defined compute task to measure. We specified the sizes by starting from a well-performing single-node specification that uses most of the memory on a single node (128MB). When run alone, this takes less than 15 minutes to complete. We then extrapolated to higher node counts by approximating the same amount of work, thus approximately preserving the total runtime of each experiment. This means that, while runs of the same node count are comparable, runs of different node counts are not directly comparable. The problem sizes we used are in Table II.

TABLE II
HPL Parameters by Node Count

Node Count	Row Count (N)	Grid P	Grid Q
1	91048	7	8
2	114713	14	8
4	144529	14	16
8	182096	28	16
16	229427	28	32
32	289059	56	32
64	364192	56	64
128	458853	112	64

2) IOR: We used IOR to approximate a data-intensive write task. We designed the IOR task to be as disruptive to object storage daemons as possible by having it create many small synchronous writes from as many processes as possible throughout the runtime of the compute tasks. Since the I/O tasks are designed to run through the entire compute task, we structured the IOR job so that it would not reasonably terminate during the computation. Once the computation was complete, the IOR task was killed. Table III describes the specific options we used for IOR.

TABLE III
IOR PARAMETERS

Parameter	Description	Value
[srun] -n	Processes (per node)	56
-t	Transfer size (bytes)	512
-T	Maximum run duration (minutes)	20
-D	Stonewalling deadline (seconds)	60
-i	Test repetitions	1048576
-e	Sync after each write phase	enabled
-C	Reorder tasks	enabled
-w	Perform write test	enabled
-a	Access method	POSIX
-s	Number of segments	1024
-F	Use file-per-process	enabled
-Y	Sync after every write	enabled

IV. RESULTS

This set of experiment included a small set of anomalies that we note here for completeness, but do not affect our analysis. All runs were completed between 7 and 10 times, with few exceptions. First, since we anticipated the "Matching Lustre" cases to be less variable, we only ran those experiments only three times each. Second, we were unable to get the largest Lustre run category (128-node HPL + 128-node IOR) completed before the deadline, as the system is a busy production platform. We will include this last data point before final publication. In the case of the 128-node HPL+BeeOND executions, we occasionally experienced runtimes slightly longer than the 20-minute IOR limit. We believe this did not significantly impact measurements, as the average runtime was less than 5% longer than 20 minutes.

Overall trends. Based on our full results, shown in Figure 5, it is clear that introducing a storage workload on BeeOND daemons running alongside HPL cause statistically significant impact to HPL runtime. Introducing a single IOR process affected jobs of all sizes, causing the HPL-only job to increase its runtime by 7-13% for 128 processes. Further increasing the I/O load with matching processes caused even greater execution times, with the 128-node Matching BeeOND (No Metadata) case experiencing 47-52% extended runtime.

Metadata server effects. In the "Beeond Matching, Skip Metadata" experiments, we were sure to keep the multi-node HPL from running on the node that hosted the metadata server. This was in contrast to the "BeeOND Matching," where the multi-node HPL would overlap with the metadata server. The intention was to separate additional metadata tasks caused by our IOR workload to be from measurement, showing whether object storage processes caused less runtime impact generally. Our experiments did not definitively demonstrate a difference runtime. Determining the extent of any impact will require further experimentation focusing on metadata workloads, as well as how different types of workloads may be balanced by BeeOND.

Overhead of idle BeeOND daemons. Figure 6 shows a detailed view of a particularly interesting phenomenon. When running HPL-only configurations, we used the same job scripts that were used for BeeOND-enabled IOR tests. Therefore, the same Slurm constraint that caused BeeOND startup were included, so those processes were running on those nodes. However, for the Lustre-specific runs, we did not specify the BeeOND constraint, so did not load any BeeOND daemons. Despite a total absence of storage operations during the HPL-only jobs, the HPL-only jobs were slower than the Lustre+IOR jobs by a statistically significant margin. For the 64-node HPL cases, this impact was likely between 0.9 and 2.5%. This impact grows with the size of the job, indicating that large scale runs could see even larger slowdowns. This is congruent with findings from previous work that showed daemon processes on Linux clusters could consume enough cycles to impact larger scale tasks [**?**]. This was a surprising finding, and we intend to pursue this line of investigation further. While the confounding factor of the Lustre IOR prevents us from definitively asserting that idle BeeOND daemons caused an overhead, a simple variation of this experiment will definitively show whether this link exists. Such an experiment will be run and reported for an accepted version of this paper.

V. DISCUSSION

In the era of massively multicore computing, where it is commonplace to see node architectures with dozens of cores, colocating system or user services with compute tasks is usually seen as a minor imposition on a running application. However, it is clear that using typical scheduling tactics and process layouts can significantly impact the runtime of applications. This has been demonstrated to be especially true at scale. How can we mitigate these impacts while continuing to offer useful services to HPC users?

Any solution compute interference should understand the requirements of the application and user expectation. While this work looks at the problem only from the perspective of compute impact, there is the related concern of I/O impact. Any compute impact mitigation has the potential to impact storage performance, which may be the more important to some users. Therefore, we encourage multiple, possibly conflicting mitigations to be made available to allow maximum flexibility for the end user.

One long-used mitigation strategy is core specialization, where some cores are dedicated to specific system services. This is becoming more pertinent with the advent of efficiency cores in modern CPUs. The main idea is that computations are pinned to some number of cores that are dedicated to their use, while other services are scheduled onto a separate subset

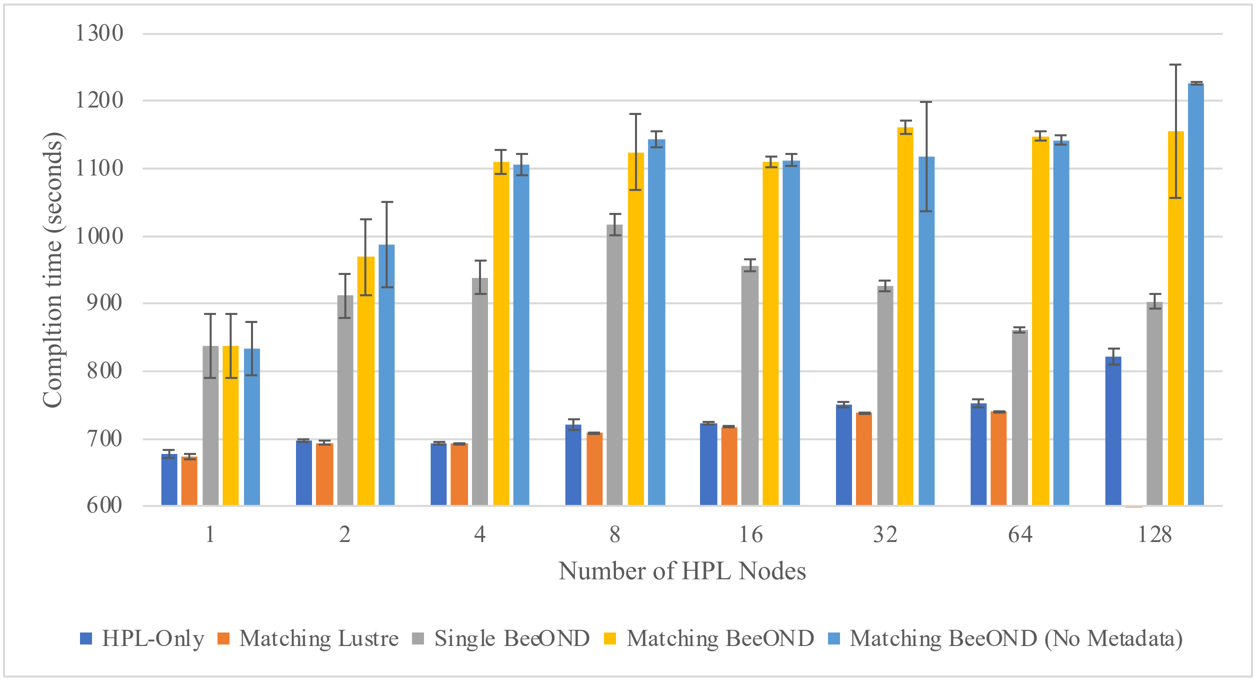


Fig. 5. Execution times of HPL with and without IOR processes co-located within the partition. Error bars indicate 95% confidence interval.

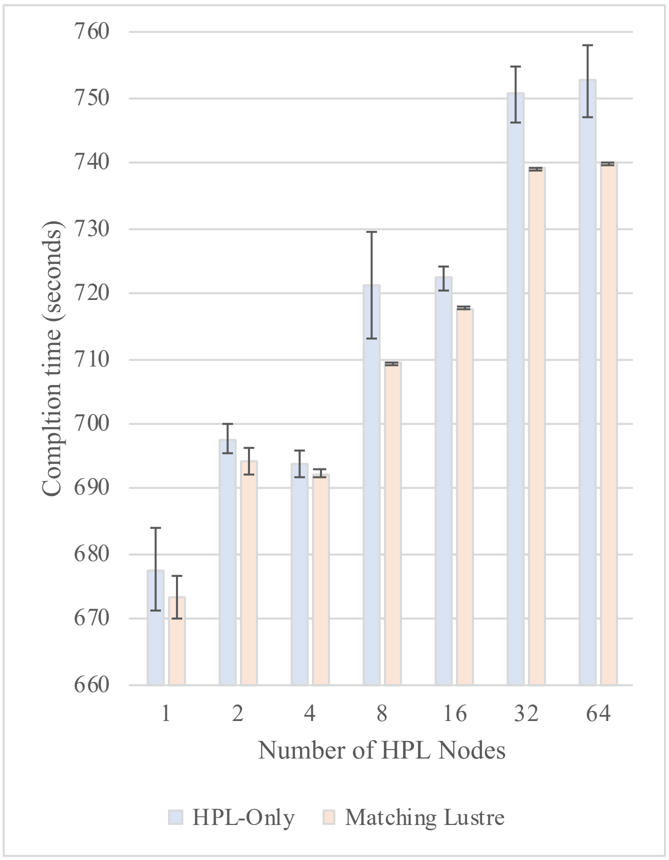


Fig. 6. A detailed view of HPL execution variance between HPL-only tasks (with BeeOND processes running) and HPL running alongside IOR targetting Lustre (without BeeOND processes running).

of cores, allowing for less interference between them. One downside of this strategy is its static nature. A job submission is typically specified as the number of processes per node, so the split between specialized cores and service cores must be decided in advance. Over-estimating or under-estimating the number of cores required for services will constrain the available performance of the system. However, using so-called "efficiency cores" for services may mitigate this somewhat, as these cores are more tuned to handing tasks like I/O, creating a natural place to draw the line between compute and services. This can work well if overheads are expected to consume at least a full core, and compute task performance is important.

In absence of core specialization, CPU and network quotas can provide another means for limiting the impact of storage on compute. Simply indicating to the operating system that storage daemons may only consume some percentage of resources would control the maximum impact of these services. Relatedly, creating a mechanism to throttle network traffic from file system clients at the source will go some distance toward making the storage system self-regulating. Unfortunately, this would do little to mitigate hot spots, where all clients are attempting to access the same small set of servers for storage services. For tightly coupled, balanced jobs, that could cause broad performance degradation.

Another strategy for ensuring minimal application impact is to allow users to control where file system processes are located within a job. In the case of the example benchmark in this paper, simply starting all BeeOND processes on nodes that were engaged in IOR runs would exempt HPL-running nodes from any impacts. A downside for typical HPC architectures with direct-attached storage is the loss of use of SSDs kept within those nodes, causing underutilization and smaller/slower file systems. While it may be possible to use network protocols such as NVMe Over Fabrics on exempt nodes to share the block storage with the nodes running daemons, the host node may still experience some runtime impacts. Less risky but more costly solutions include architectures with disaggregated storage, designed as Just-a-Bunch-Of-Flash chassis. One can also simply expand the size of the job to include extra servers specifically for file system services, if the user anticipates high utilization.

VI. Conclusions and Future Work

In this paper, we reviewed a set of experiments where we combined IOR, a storage benchmark, against HPL, a compute benchmark simulating a tightly coupled linear algebra application. We ran these benchmarks on a system with per-allocation BeeOND file system capabilities, which gives a user a private file system by running storage services on compute nodes alongside running applications. Our goal was to show whether heavy storage use could impact compute task performance.

We were able to show that, even for large compute jobs (128 nodes, 7168 cores) paired with a small number of I/O processes (1 node, 56 processes), there was significant impact to the HPL runtimes. In some cases, HPL runtime was increased by up to 52%. Surprisingly, we were also able to measure a potential impact (up to 2.5% on 64 nodes) from simply running BeeOND processes alongside HPL without any I/O traffic. Such a phenomenon indicates that running BeeOND daemons on dedicated cores is a reasonable strategy as job node counts increase.

This research has uncovered a large number of future research directions. While we have uncovered CPU-based performance impacts from I/O daemons, we did not receive clear results regarding impact of metadata services. This is likely because the IOR benchmark parameters we used did not heavily exercise the metadata facilities of BeeOND. We intend to design new experiments that will specifically exercise metadata operations. We also intend to explore user-controlled placement of BeeOND processes to mitigate compute performance impact.