Hybrid Resource Management for HPC and Data Intensive Workloads*

Abel Souza,[1] Mohamad Rezaei,[2] Erwin Laure,[2] and Johan Tordsson[1]

[1]Department of Computing Science
Umeå University, Sweden
{abel,tordsson}@cs.umu.se

[2]PDC Center for High Performance Computing
KTH Royal Institute of Technology, Sweden
{mrez,erwinl}@kth.se

Abstract: High Performance Computing (HPC) and Data Intensive (DI) workloads have been executed on separate clusters using different tools for resource and application management. With increasing convergence, where modern applications are composed of both types of jobs in complex workflows, this separation becomes a growing overhead and the need for a common platform increases. Executing both workload classes on the same clusters not only enables hybrid workflows, but can also increase system efficiency, as available hardware often is not fully utilized by applications. While HPC systems are typically managed in a coarse grained fashion, with exclusive resource allocations, DI systems employ a finer grained regime, enabling dynamic allocation and control based on application needs. On the path to full convergence, a useful and less intrusive step is a hybrid resource management system allowing the execution of DI applications on top of standard HPC scheduling systems. In this paper we present the architecture of a hybrid system enabling dual-level scheduling for DI jobs in HPC infrastructures. Our system takes advantage of real-time resource profiling to efficiently co-schedule HPC and DI applications. The architecture is easily extensible to current and new types of distributed applications, allowing efficient combination of hybrid workloads on HPC resources with increased job throughput and higher overall resource utilization. The implementation is based on the Slurm and Mesos resource managers for HPC and DI jobs. Experimental evaluations in a real cluster based on a set of representative HPC and DI applications demonstrate that our hybrid architecture improves resource utilization by 20%, with 12% decrease on queue makespan while still meeting all deadlines for HPC jobs.

Key words: Resource Management, High Performance Computing, Data Intensive Computing, Mesos, Slurm, Bootstrapping

*The paper has been re-typeset to match the thesis style. Reproduced with permission of IEEE.

1 Introduction

The increasing convergence of High Performance Computing (HPC) and Data Intensive (DI) applications calls for using a single infrastructure, not only to better utilize the ever increasing power of computing hardware but also to enable complex workflows, combining HPC and DI. While traditionally separate infrastructures have been used for HPC and DI applications, modern systems not only have tremendous compute power on a single node (e.g. over 40 TFLOPS on Summit, a recent Oak Ridge Leadership Computing Facility supercomputer system [Hin18]) but also enormous memory and storage capabilities provided for instance via high bandwidth memory and local Non-Volatile Random Access Memory (NVRAM) storage, making them efficient tools for both types of workloads. Still, the usage models and software stacks for these applications are very different. While HPC applications are long-running jobs, parallelized with tools like MPI or OpenMP, with static resource allocations and an a-priori determined lifespan, DI applications exploit models like MapReduce [DG08] and make use of frameworks such as Hadoop [Whi12] and Spark [Zah+10], which require dynamic resource allocations to adapt to changing compute requirements and also changes in hardware availability, making them *adaptive jobs* [Pra+15], also known as *reactive applications*. However, HPC resource managers like Slurm [YJG03] or Torque [Sta06] assign resources to jobs based on what has been specified in their requests. Jobs can typically only change their allocations by cancelling and resubmitting to the queue and not adjust during runtime [Reu+18]. On the other hand, DI schedulers, like YARN or Mesos [Vav+13; Hin+11] are designed to provide low-latency, dynamic allocations [Jha+14]. Hence, there is a need for convergence of these resource allocation models to fully support hybrid application workflows. This convergence will not only enable DI applications on HPC systems but is also needed for the evolution of traditional HPC applications that are increasingly employing complex workflows, in-situ data processing, and dynamic restructuring, e.g. mesh refinement. These characteristics make HPC jobs structurally similar to DI jobs [AW17; Tiw+13; Reu+16; Reu+18; Com+16a; Ber17].

Furthermore, also at runtime much care needs to be taken when co-scheduling HPC and DI applications [Bre+12]. HPC applications typically do not fully utilize all the resources allocated to them, and this is likely to get worse with the increased performance of compute nodes like the ones mentioned above. These characteristics offer the potential to allocate underutilized resources to other applications, for instance, DI jobs [Mer+17]. Furthermore, HPC applications are typically very sensitive to disturbance in resource usage, such as CPU, memory, network bandwidth and/or Disk Input/Output (I/O) operations [Bha+13]. Hence, naively co-scheduling HPC and DI applications on shared resources will inevitably reduce the efficiency of HPC jobs. Fortunately, HPC applications have a repeating nature, where the same application is typically executed many times with different input data, with predictable resource usage patterns (e.g., CPU, memory, I/O, network) based on monitoring and profiling. This information can be analyzed through statistical learning techniques, enabling policies aimed at improving queue throughput and resource utilization [Eme+13; RS00] with controlled performance impacts to applications. Figure 1 shows such a typical HPC

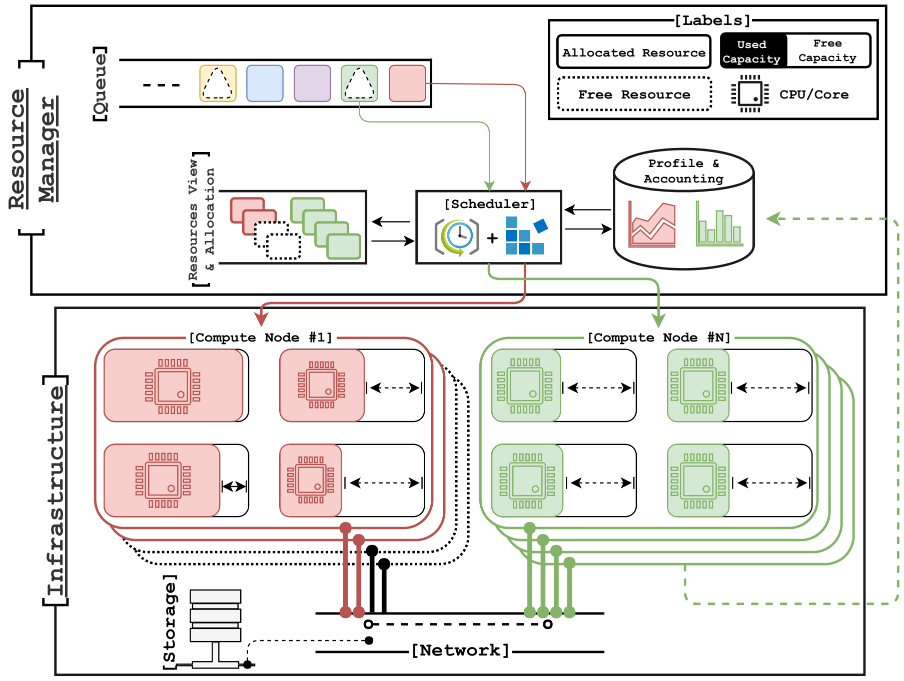


Figure 1: Typical Resource Management in a HPC cluster. Space sharing: Jobs with different (resource) characteristics are queued and allocated resources (compute nodes in the infrastructure) when they become free.

resource manager and monitoring system: (colored) jobs are queued, and requested resources allocated in the compute nodes by the scheduler (big red and green parallel rectangles in the infrastructure). During job execution, resources' capacity utilization and idleness can be monitored and profiled (white areas within compute nodes).

In this paper we present an architecture for co-scheduling HPC and DI application based on resource profile and usage predictions for dynamic throttling of DI applications. Integrating more dynamic features into existing HPC schedulers like Slurm is however an intrusive task, and experimental schedulers are difficult to deploy on production HPC systems. We thus adopt a hybrid approach, where HPC applications are still scheduled using a standard HPC scheduler, but co-scheduling of DI applications is done through Mesos. DI jobs scheduling is based on resource usage estimates and DI jobs are dynamically throttled when they interfere with HPC applications. With this, we increase overall resource usage and efficiency as well as provide a path for building complex workflows that combines HPC and DI components.

The remainder of this paper is organized as follows. Section 2 discusses notable past references on scheduling and resource management in HPC infrastructures, and what changes would be needed to extend them. Section 3 describes our approach in detail. Results and discussion follow in Sections 4 and 5. Conclusions and future work go on Section 6.

2 Background & Related Work

HPC clusters are managed by resource managers like Slurm [YJG03] and Torque [Sta06], which commonly require users to describe jobs by the total run time (also called deadline) of the allocation and a geometry (i.e., number of CPUs, threads per core, total memory, specific accelerators and/or network bandwidth/topology). Jobs can finish sooner than the specified run time limits, but more importantly, can utilize less than the total resource capacity allocated. The resource allocation to jobs is commonly at the granularity of a node. Some resource managers allow nodes to be shared between jobs, although this feature is often not enabled by default. Having resource allocation granularity set to node level guarantees predictable performance (capacity), which is needed for various purposes like cache optimization, debugging, and is thus important during application development and testing. Usually, users cannot request additional resources other than the ones already allocated at job launch [Amv+17], even though many libraries vow in this direction [Com+16b]. Job categories describe what can happen to a job's resource geometry throughout its life-cycle. Historically, HPC jobs have been categorized in four types: rigid, moldable, malleable, and evolving [FR96]. With large data processing needs becoming a norm both in industry and in scientific communities, a fifth class has appeared: *adaptive* jobs [Pra+15]. These jobs are characterized by handling large amounts of data, being highly dynamic and adaptable to resource changes, faults, and by being very data-intensive [Pra+15; Reu+18]. Thus, they are also referred to as Data Intensive (DI) jobs/applications.

Large parts of HPC datacenters are reserved to rigid and long running workflows [Amv+17; Reu+18] that require predictable reservations, where users specify resource constraints in very detailed manner before job submission. HPC jobs come with several constraints as they are tightly coupled in nature, requiring periodical message passing, synchronization barriers [Val90], and checkpointing for fault-tolerance [Wan+10]. These datacenters use a centralized scheduling and queueing system (Figure 1), and jobs do not have the same latency requirements as DI jobs [Pra+15], with longer waiting-times reported [Amv+17].

2.1 Dynamic Resource Managers

Resource assignment can be a challenging task in clusters with heterogeneous resources, where compute nodes with different configurations and architectures are used. For heterogeneous environments, dynamic resource managers are used as they are able to cope with variations and faults within the infrastructure [Reu+18]. In traditional HPC resource managers, allocation is the assignment of resources to execute a job. This means the request description is what the resource manager will allocate to the job. This is the common Service Level Agreement (SLA) that most HPC clusters support. Collocation (also known as co-scheduling) is a common technique to increase resource utilization in clusters, though operators are reluctant to use this due to the potential performance interference caused by node sharing, known as SLA violations. For instance, to mediate performance interference Paragon [DK13] uses classification to weight the impact of different resources for each job, and uses this knowledge to select candidates for collocating jobs. On-line models are also used to detect and avoid performance interference [NK10; Yan+13], or to take actions such as throttling low-priority jobs to mend the interference [Zha+13]. Off-line models can be used as well [Sha+13], but do not help at runtime. Other resource managers like Mesos [Hin+11], Torque, Omega and Kubernetes [Sch+13] expect workloads to request resource reservations. Mesos, for instance, processes resource requests and, based on availability and fairness, makes resource offers to individual frameworks (e.g., Hadoop), which can accept or reject offers depending on application requirements. Mesos simplifies heterogeneity by behaving like a meta-scheduler, with conceptual abstractions for CPU, memory and other resources, which are taken away from physical nodes and managed by Mesos. This enables a set of new capabilities like isolation, elasticity and fault-tolerance for distributed applications [Reu+18].

With finer granularity in task allocation, one can expect higher resource utilization [Zha+13; Reu+18], but in large clusters this can have negative impacts due to the lower resource fragmentation. Thus, a method for enforcing resource isolation among jobs is essential [Zha+13; Bur+16]. Although HPC resource managers like Slurm can allocate resources using finer granularities, they do not provide the necessary application programming interface (API) and capabilities for application elasticity at runtime, nor mechanisms for controlling and enforcing isolation between jobs and tasks [Hin+09]. The main difference between dynamic (Mesos or Yarn [Vav+13]) and static resource managers (Slurm or Torque) is this extra API providing fault-tolerance, execution and resource control at runtime [Reu+18]. For example, Mesos provides APIs that use cgroups [Men07] as its underlying resource isolation mechanism. In Linux, cgroups is one of the most available and robust fine grained operating system controls that make sure processes, encapsulated as containers (namespaces), do not consume more of the resource capacity (e.g. CPU, memory, I/O and/or network) than what has been assigned to them. These capabilities are important for load-balancing and stream-processing that rely on task migration and/or resource allocation changes during runtime [Hin+09].

2.2 Hybrid Resource Manager Challenges

In particular for large scale HPC clusters, having a hybrid resource manager combining static and dynamic management has a number of advantages, as well as challenges. First, HPC applications tend to have higher resource utilization than cloud computing applications [Amv+17], which requires any combined management solution to be scalable in the number of utilized resources. Secondly, as DI applications require low-latency scheduling, the scheduling needs to be performed quickly and in real-time to make resources available. Tools presented in [Eva+14] and [Pal+15] allow users to analyze and optimize their applications, though they do not enable the integration of collected information for real-time scheduling and resource control. Lastly, the resource manager should be able to handle various criteria in parallel. For instance, resource allocation must be performed in a manner that also considers isolation and interference mitigation. Large scale HPC clusters on their way to Exascale computing are going to grow in both resource size and heterogeneity. In addition, DI applications are becoming more recurrent, complex and diverse[Ous+13]. To address these diversity of requirements, dynamic resource managers provide a number of APIs for elasticity, migration and fault-tolerance [Reu+18]. However, the common practice in HPC clusters is to statically allocate resources to jobs, which means that at runtime a job cannot request for changes in its allocation nor the resource manager can change an allocation without ending a job's execution.

Predicting system utilization of parallel jobs have been studied extensively [NK10; Mar+11; DK14; Yan+13; BTG13], but adding certainty (for example, confidence intervals) to these predictions have not been prevalent. One of the main focuses of our design is to enable a resource manager to take decisions with confidence. The confidence can be used to lower occurrences of false positives (incorrect collocations resulting in notable performance interference) while sharing resources, which is essential as performance is to be prioritized to HPC applications. Furthermore, any combination of static and dynamic resource managers must be simple and scalable, and must detect and gracefully deal with interference. In here, we propose, evaluate and discuss the design of a non-intrusive hybrid architecture combining traditional HPC scheduling with the advantages of more dynamic approaches taken by DI schedulers. The goal is to improve DI application scheduling, overall datacenter resource usage, queue waiting time, and queue makespan by deriving spare resources from traditional HPC job allocations using data analytics techniques. In here, resource usage refers to how efficiently the capacity of the allocated resources are actually used by the application. Queue waiting time and makespan respectively relate to how long a job waits until it starts executing and how long the batch processing system (e.g., Slurm) takes to complete a given set of jobs in a queue instance. Significant amount of research has been done on resource management in distributed systems with focus on requirements of HPC or DI applications. The main agenda of this work is to utilize HPC infrastructures for DI applications with their contradictory SLAs versus the common approach HPC resource managers have toward running and managing jobs. In the following we structure various main points which enable us to design our proposed architecture.

3 Architecture

In this section we describe the design of our hybrid architecture. Our aim is to collocate HPC and DI jobs in order to increase overall cluster utilization, with controlled performance overheads for HPC jobs. As discussed in previous sections, to achieve full convergence and considerable performance for HPC and DI jobs, intrusive changes in current HPC infrastructures would be needed. However, in here we present a hybrid, *non-intrusive* approach focused on two resource managers: Slurm and Mesos. Mesos is chosen as it is non-intrusive and can be set and executed by regular users. In contrast, incorporating the very popular Kubernetes [Sch+13] resource manager in a HPC cluster would require administrator privileges. Figure 2 shows the main components of our hybrid plugin architecture and exemplifies the overall job life-cycle of a queue managed by Slurm. In the *Job Queue*, full colored squares represent traditional HPC jobs and squares with a full triangle inside denote DI jobs. These jobs could possibly be tagged by users during job submission or submitted to special queues. The blue arrows, originating from the first job and ordered from top to bottom, show each step taken throughout a job's life-cycle, from submission to spare resource inferring in the *Insight Engine* (see next sub-section). Unlike in the traditional HPC scheduling scenario (Figure 1), DI applications are sent to application-dependant queues managed by specific frameworks inside Mesos. Specific job allocations are coloured according to the job's color (Blue and green boxes in the *Infrastructure* layer). By default, a Slurm daemon (SLURMD) runs in each node. In addition, our architecture spawns additional daemons (Mesos agents) on each node for communication with Mesos. Current resource utilization in each node is exemplified by the arrows in the squares and rectangles, which represent estimations for CPU core and memory usage, respectively.

Mesos performs resource management and scheduling across an entire datacenter and handles all communications with application schedulers through an API. This API enables scheduling frameworks to monitor and execute tasks spawned by applications. A framework in Mesos is the implementation of a scheduler tailored specifically to an application-type (e.g. Spark or Storm). In Figure 2, frameworks are represented by rectangles on top of the Mesos Master (MapReduce and DataFlow schedulers). Mesos monitors task states (`RUN, COMPLETE, FAIL,` etc.) to handle problems such as application crashes, misbehaviours, or unresponsiveness. For each case the frameworks can trigger specific actions, e.g., restarting or migrating a failed task to another node by communicating with the master. This model and the associated runtime system enable applications with enhancements such as fault-tolerance, resource isolation, and performance control.

3.1 Insight Engine

The overall application performance profile is bootstrapped by the *Insight Engine* (IE, far left box inside the *Hybrid Plugin* in Figure 2). The IE infers spare resources through statistical analysis and its internal process is depicted in Figure 3 and Algorithm 1 (see next subsection). The IE calculates and generates a cluster-wide resource capacity view, with a database (not shown) from where jobs' resource utilization is shared with Mesos through its *Allocator Module*. Spare resources are periodically calculated by the IE and registered with the Allocator Module. Such resources can be offered to the frameworks via the Master to allow applications to start execution. The default way for Mesos to allocate resources to frameworks is based on the Dominant Resource Fairness (DRF) algorithm [Gho+11; Hin+11]. In essence, Mesos frameworks receive offers from the scheduler and then decide if they suit their specific applications resource needs. This model has been shown to cause starvation, but Mesos mitigates this problem by supporting weighted DRF among frameworks [PPS15].

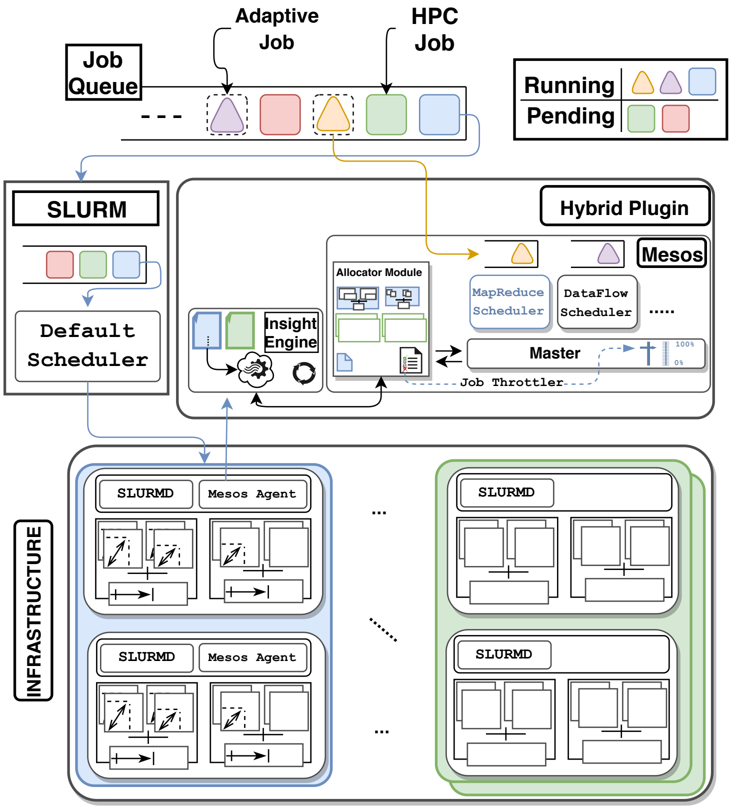


Figure 2: Hybrid Architecture with job queue and outline of the job submission life-cycle. Blue arrows show job allocation and profile, with the Insight Engine inferring spare resources that later that can be offered to application-specific schedulers (MapReduce/DataFlow) sitting on top of the Master. Spare resources (in the green allocation) are pending.

The profiling process starts after a set of nodes are allocated to a HPC job submitted through Slurm (”Job Queue” in Figure 2). The moment the job tasks start execution, a profiler starts collecting the selected performance counters (performed through Slurm Prolog scripts). Performance counters are CPU hardware registers that count events such as CPU utilization, CPI (cycles per instruction), cache-misses, or branch miss-predictions. These metrics form a basis for profiling applications to trace dynamic control flow and identify any hot spots. The metrics are sent from allocated nodes to the IE, which then estimates resource utilization within the corresponding (tunable) confidence intervals. Extensive collection of performance counters can negatively impact a job’s performance. We thus choose to use *perf* [Mel09] for instrumenting CPU performance counters as it has a very lightweight profile footprint [Wea13]. Perf is also included in the Linux kernel and is frequently updated and enhanced.

In order to infer free resources and potential task interference, the IE needs to analyze data from all running tasks and cluster nodes, potentially a very large dataset. The IE deals with the volume of data by using the Bags of Little Bootstraps method[Kle+12] which provides the results in a limited time. Using BLB, the IE can look into significantly smaller portion of performance data while providing results within a preset time constraint. BLB also provides a simple and robust method of assessing the quality of estimations during operations. We thus use a scalable version of bootstrapping for time series to enable the resource manager's scheduler to create resource usage estimations in a scalable and robust manner. BLB works by averaging the results of

```
Data: Performance Counter Traces
Result: Node $i$'s Utilization Estimates with Confidence
$R_i \leftarrow perf$ ;                                   ▷ data stream from node $i$
while $triggered$ do
    $k \leftarrow 1$
    while $\xi_i^* < \xi_{confidence}^i$ do
        $S^* \leftarrow s$ disjoint samples from $R_i$ of size $b$
        for each $j \leftarrow 1$ to $s$ do
            for $l \leftarrow 1$ to $k$ do
                $N_b^j \leftarrow$ Resample $S^*$, $n$ times ($= |R_i|$)
                $\mathbb{P}_l^* \leftarrow n^{-1} \sum_{a=1}^{b} N_a^j \delta_{R_i,a}$ ;  ▷ Empirical distribution
                 for each $N^j$
                $\theta_l^* \leftarrow \theta(\mathbb{P}_l^*)$ ;                  ▷ Mean of $\mathbb{P}_l^*$
            end
            $Q_{k,j}^* \leftarrow k^{-1} \sum_{l=1}^{k} \delta_{\theta_l^*}$ ;            ▷ Empirical mean
            $\xi_i^* \leftarrow \xi(Q_{k,j}^*)$ ;                     ▷ Empirical confidence
        end
        $k \leftarrow k+1$
    end
    $Q^i \leftarrow s^{-1} \sum_{j=1}^{s} Q_{k,j}^*$
    $\xi^i \leftarrow s^{-1} \sum_{j=1}^{s} \xi_j^*$
    $return(Q^i, \xi^i)$
end
```

Algorithm 1: Upon invocation, the IE runs BLB for each node i to create the needed resource estimates for the selected job and returns the results: Q^i, for mean resource usage, and ξ^i, for the confidence interval. These values are used for the job's soft (Q^i) and hard limits (Q^i + ξ^i) for each collected (resource) metric, respectively.

bootstrapping multiple disjoint subsets of a large dataset of size n. Broadly speaking, BLB uniformly samples s considerable small (of size b; such that $b < n$) subsets from the dataset. BLB creates an empirical (δ_R) distribution for each subset s, and a mean and confidence are estimated in the manner of classic bootstraping: BLB repeatedly resamples (k times) n points (independent and identically distributed, i.i.d.) and creates a distribution associated to each subset term (j), averaging their mean values. From each associated empirical distribution, it computes the mean, and approximates the confidence by averaging all confidences from every Q_j^*. Algorithm 1 basically looks into a data stream of performance counters for the selected node i, and then creates its samples with replacement by sampling a growing dataset of 1 minute intervals in a time series. This method is repeated until node i's mean confidence (ξ_i^*) is below a preset threshold ($\xi_{confidence}^i$), associated with node i. Perfect time estimates are not required, but reasonable approximations are valuable for resource managers as one of their aim is to improve overall (datacenter) utilization. With the goal of having minimal computational overhead on the local machine, performance counters are sent to the Mesos Master, located outside the scope of the job's resource allocation, and the statistics calculated are limited to the mean and standard deviation for each metric. For the purpose of simplicity, in this paper we only focus on CPU utilization, represented as the *cpu-clock* counter in perf. As shown in Algorithm 1, we calculate the uncertainty of these measurements for window size W until we reach a confidence of $\xi_{confidence,j}$, empirically chosen as 80% for all nodes, based on our HPC use-cases (see Section 4). Note that $\xi_{confidence}^i$ confidence could be set on a per node basis as well, although we do not analyze this feature in here.

Note that the gathered data from each node could be combined with a Gaussian distribution, which would enable the IE to estimate utilization's mean and standard deviation (confidence) for the whole job. This latter information can be used for planning job co-scheduling. The balance between the sample rate, the number of re-sampling, and the predefined confidence target can define the latency with which BLB provides the new estimates in each job. In the end we get the estimations from only small portion of the whole node's dataset [Kle+12], which saves computations, storage, network communication, and reaction time (when a job changes resource usage behaviour). More importantly, the whole pipeline of creating the insight (both estimation and confidence) are scalable by design. We should also consider that for the performance data we can set a retention time in which samples will be taken from. Any job running in the cluster can change its behavior during execution and the retention time can be used to reduce false assumptions about the current state of the job. In this work we empirically set the best confidence that works for all the test applications (0.8 in Algorithm 1), and keep that as a constant value for each job's node j. For future work we consider using other means to set the best confidence, ideally in a per application basis. This confidence can have direct impacts on false positives (interference).

Advantages of this design include that the central IE is lightweight as the main part of decision process is distributed. More importantly, even though HPC jobs are expected to be load-balanced across nodes, some jobs may not or different tasks of the same job can have different resource usage patterns. In this case, Algorithm 1 reports different resource utilization (as mean and confidence) for each node. The confidence is used to show when the measures are going to be useful for the resource manager to collocate and control DI jobs resource shares. In essence, when Algorithm 1 reaches 80 percent confidence it updates the values to Mesos master with one caveat - each

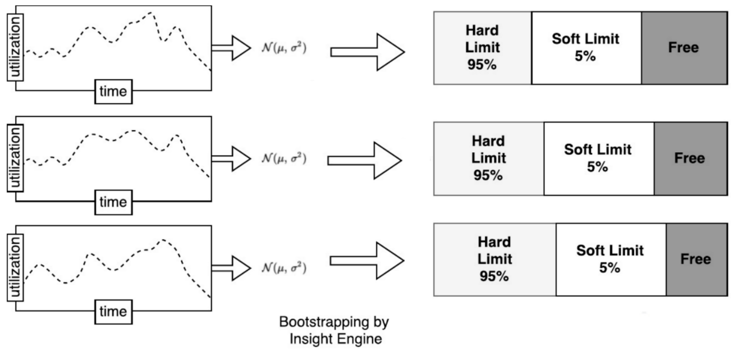


Figure 3: From model's estimation to cgroup limits: For each node of a job we use the confidence set as the standard deviation added to the mean utilization. Statistically this number defines the expected range for the node utilization.

local node defines hard and soft resource limits for DI jobs based on the the mean and confidence interval. These limits are enforced using cgroups (see Subsection 3.B).

3.1.1 Confidence

The Hybrid Resource Manager needs to have confident knowledge of its long running jobs resource utilization. By using BLB, the IE creates a distribution for the job's various metrics (performance counters) with the added benefit of a confidence score for this estimate (we assume the distribution to be Gaussian). We use the confidence to determine if the expectation of a job's behavior (e.g. in form of an estimation) is stable enough to be usable, but also to reduce occurrences of false positives. As our hybrid approach targets the HPC domain, the resource manager has a priority of not disturbing the execution of HPC jobs. Here, the confidence helps in cases where HPC jobs' resource utilization sporadically changes. Another usage for the confidence is to identify when the classification of a job (e.g. CPU, memory, I/O, or network bounded) changes, or put it another way, to analyze when the spectrum of job's utilization as we expect it to be, changes (Figure 3).

3.1.2 Scalability

Another important aspect of HPC management relates to how scalable the IE model is. In large HPC clusters, massive amounts of data can by produced by a single job, by for instance, increasing or decreasing the frequency of gathered performance counters. A HPC job generally produces a correlated stream of data between its allocated nodes due to load-balacing, a desired quality that reduces time to complete, although due to sharing this not always hold. BLB helps with reducing the amount of data that needs to be read by the IE model. In here we do not assume that HPC applications have correlated resource utilization among allocated nodes.Another source of scalability is in the variations in expectations. We expect models to become more stable (higher confidence) over a period of time. In case of false positives which can be a result of interference, or just an interval model update, the IE can be updated based on the current performance data. But unless the job erratically changes its behavior we would not expect these updates to happen very often, and even they do occur, they only make the model's expectation less confident. This in term leads to the IE not collocating any other jobs in the affected nodes.

3.1.3 Real-Time Decisions

As shown in Algorithm 1, the first sampling without replacement is being done based on number of tasks of a job (Commonly, number of allocated nodes to the job). The algorithm performs the next step of iterative estimations until a certain confidence is reached. This method can be altered by setting a time limit to terminate the iteration. This approach is normally robust and could be run periodically, but at times outliers may occur as the job changes its behavior sporadically. More importantly, DI jobs can be reactive and Algorithm 1 may be restarted reactively (on-demand). The key point is that Algorithm 1 is bounded by either confidence or time limits, while still being scalable.

3.2 Sharing & Isolation: Job Throttler

There are various ways to enforce isolation between collocated jobs in the same node. One is through the operating system's (OS) scheduler, which may not provide enough guarantees for memory operations due to the impacts of context switches in the Last Level Cache (LLC) [Zha+13]. Another way is by using a monitoring agent on the nodes where jobs are running, which is a very promising approach [Sch+13; Zha+13], however it works only on specific architectures and the resource manager would need to be aware of different hardware. Linux Cgroups [Men07] isolation is one of the most robust ways to ensure that processes do not consume more resources than what has been assigned to them. The LLC problem can be addressed by cgroups in some processors, a solution already deployed in some datacenters [Her+16]. The main issue with cgroups is that within the context of a container the task might not use all the resources. We address this by use of both hard and soft limits in cgroup, which to date is not yet a well-established feature.

Job throttling aims at reducing CPU access to applications at the cost of computational performance. Mesos uses cgroups to provide resource isolation for CPU, memory, I/O and network bandwidth. Resources received by Mesos frameworks to execute applications are controlled by the Allocator Module through Mesos APIs. Although all isolation mechanisms are provided by Mesos, performance interference due to hardware space-sharing among different applications can still occur and impact collocated jobs negatively. It is well know that frequently collecting hardware metrics can have negative impacts on the workload. For the purpose of our approach, we use CPU utilization (*cpu-clock* in perf metrics) together with our algorithm to detect variations in higher priority, HPC workloads. Other ways to achieve this is to collect information directly from within the application (e.g. throughput or latency overtime), but this would require changes to runtime systems and libraries, which reduces the potential for adoption of the solution.

4 Evaluation

In this section, we evaluate our proposed architecture with respect to performance, resource utilization, and queue throughput for different applications with different resource usage profiles and scenarios. In order to demonstrate the feasibility of our hybrid approach, we first run two HPC applications and compare isolated runs with statically and dynamically controlled scenarios. We then use a workload model to generate a job trace derived from the Argonne National Laboratory supercomputer. The generator was taken from the Parallel Archive Workload [FTK14]. This is a comprehensive and realistic model for generating streams of rigid jobs with variable geometries following a given cluster's size [CB01]. The same model is used for three different cluster sizes, measured in number of cores: 128, 256, and 512 cores. Time allocations were granted for the hour/half-hour depending on the applications' execution times when running in isolation (no node sharing). For instance, if an HPC job takes 39 minutes to complete, the job is submitted as if the user requested 1 hour. We evaluated the impact of our approach in a private and dedicated cluster, hosted at RISE SICS North Infrastructure and Cloud datacenter research Environment (ICE)[1]. The cluster is composed of 16 Open Compute Windmill compute nodes, each containing two 16 Intel Xeon E5-2660 CPU-cores (2.2 GHz), with 144 GB DDR4 memory (2133 MHz). All nodes are connected with a 10 Gb/s Ethernet network. The shared filesystem used during workload executions is a NFS v4.2, connected through Remote Direct Access Memory over Converged Network (RoCE), with peak performance of approximately 10 Gb/s. The cluster runs Ubuntu Bionic Beaver (18.04), and is managed by Slurm (17.02).

4.1 Applications

We selected five workflows as use-cases, four real ones (NEK5000, Montage, BLAST and Statistics), and one synthetic. For our evaluation, NEK5000, Montage, and BLAST are examples of HPC jobs (scientific workflows), and Statistics and Synthetic are examples of DI jobs. Scientific workflows describe applications' resource requirements in each part of the execution (called stage). Stages can be parallel or sequential. In parallel stages, all available cores are used, not necessarily in its maximum capacity. In sequential stages, only one of available cores are used, often at maximum capacity, whereas all other available cores idle. This property makes workflows a suitable candidate for collocation with DI jobs.

NEK5000 [Fis+08] is a fluid dynamics application. It solves the unsteady incompressible two-dimensional, asymmetric, or three-dimensional Stokes or Navier-Stokes

[1]https://ice.sics.se/

[Pat84] equations with forced or natural convection heat transfer in both stationary (fixed) or time-dependent geometry. NEK5000 is memory and cache-optimized and thus very CPU, as well as network intensive when scaling it (i.e., adding more nodes/resources). All runs use workloads from the standard examples that come with NEK5000.

Montage [18] is an I/O intensive workload that constructs a JPEG image from sky survey data formatted as Flexible Image Transport System (FITS) files. Montage is composed of nine stages logically grouped into parallel and sequential stages. All experimental runs of Montage construct the image for survey `M17` on `band j` and `degree 8.0` from `2mass` Atlas images.

BLAST [Alt+97] is a CPU-intensive workflow that matches DNA sequences against a large sequence database (> 6 GB). The workflow splits an input file (of few KBs) into several small files and then uses parallel tasks to compare the input against the large sequence database. The database is loaded in-memory on all compute nodes during the parallel stage. Finally, all the outputs from the parallel stage are merged into a single file. BLAST is composed of two main stages: one parallel and one sequential.

Statistics is an I/O intensive application that calculates various statistical metrics (mean, median, average, standard deviation, variance, etc.) from a big dataset with measurements of electric power consumption in one household with a one-minute sampling rate over a period of almost 4 years. Different electrical quantities and some sub-metering values are available in a public dataset [KJ11]. The statistics workflow is composed mainly of a small sequential stage and a big parallel stage, consuming most of the processing due to communication among the parallel tasks.

Synthetic workflow is composed of basic sequence and a parallel stages, written in Python. This is memory and CPU intensive application, where the memory intensive part consists of tasks that perform a large number of memory allocations for over one billion integers, prior to a CPU intensive part that calculates the values of their sum and multiplication. The first stage contains one billion tasks, calculating the sum in sequence, whereas the second parallel stage contains five million tasks, calculating the multiplication in parallel.

The workload model is used to generate 16, 28 and 41 job arrivals respectively for each cluster size: 128, 256 and 512 cores. From these, 52% of the jobs (52) are set as HPC jobs and 33 as DI jobs. HPC jobs run Montage and NEK5000 with 32, 64 and 128 cores (maximum scale), as well as BLAST with 32, 64, 128, 256, and 512 cores. The core allocation for DI jobs varies as 32 to 128 cores for Statistics and 32 to 512 cores for Synthetic.

4.2 Scenarios

We first describe an isolated experiment illustrating the performance isolation and enforcement capabilities of our approach. This experiment was performed on a single node with only two applications, Montage and BLAST, where Montage was given higher priority than BLAST. The number of cores used by each application was either 8, 16, or 32 each (three scenarios in total). We begin by running each application in isolation to establish a baseline, and later collocate the two applications, either with no resource sharing control, a static sharing by cgroups (different percentages of *cgroup.cpu_quota* allocated to Montage and BLAST), or an active sharing approach. For the latter, we compare our BLB method (Algorithm 1) with two simple methods, a parallel freezer that halts the lower priority application as long as the high priority one is using all its cores, and a random freezer that halts the lower priority application for a random amount of time whenever the high priority application is using all its cores.

We later validate our results against running the same workload generated by the model in three different scenarios: (i) vanilla Slurm, (ii) collocated Slurm, and (iii) with our hybrid architecture. In (i), applications are submitted as separated jobs, with default backfilling scheduling algorithm used by Slurm, where the granularity is at the node level. In scenario (ii), jobs are submitted as separated jobs just as in the previous case, with a virtual cluster composed of the physical nodes. This virtual cluster is managed by an additional DI queue, and its resource granularity/affinity is set at the core level (*CR_Core* Slurm setting), so nodes can be shared among jobs, with a maximum of two jobs per consumable resource (node). In scenario (ii), no two HPC jobs are collocated. In scenario (iii), our hybrid approach, a similar resource sharing occurs, but our plugin controls how resources are shared among HPC and DI jobs. As explained in the previous section (Section 3), DI jobs do not go into the normal Slurm priority queue. Instead, they are sent to Mesos queue and the application scheduler (Mesos framework) takes care of allocating and monitoring resources given to the tasks. In order to avoid DI job starvation, a minimum of 5% of every resource in the cluster is given to Mesos. This means that if collocated with a HPC job, DI jobs will be able to use at least 5% of the processing power available in the node. After collocation, the IE estimates and updates resource limits (using Algorithm 1), which keeps running in real-time during all experiments. These estimations are then used to control and enforce the resource shares for each job, where the priority are the HPC applications.

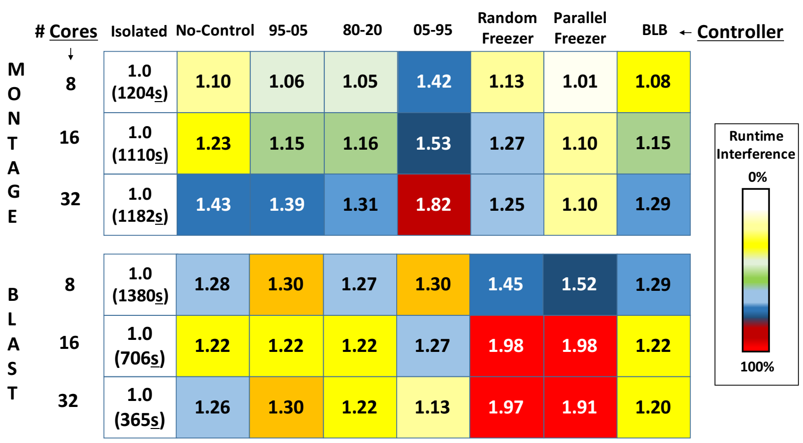
For each scenario, we collect the following metrics:

1. total queue latency, which is the time taken to complete the execution of all queued jobs;
2. CPU utilization over time for each node managed by the job queue;
3. total execution time (wall clock) for each submitted job.

These metrics can be collected from Slurm logs, and/or obtained using the *perf* tool. Notably, the use of *perf* does not seem to cause any impact on the workload, as it is already loaded directly into the Linux kernel.

4.3 Results

Starting with our isolated experiment, Figure 4 shows a heatmap with execution times for each application, normalized against the scenario where the applications execute


Runtime Interference
0%
100%
← Controller

	# Cores	Isolated	No-Control	95-05	80-20	05-95	Random Freezer	Parallel Freezer	BLB
MONTAGE	8	1.0 (1204s)	1.10	1.06	1.05	1.42	1.13	1.01	1.08
	16	1.0 (1110s)	1.23	1.15	1.16	1.53	1.27	1.10	1.15
	32	1.0 (1182s)	1.43	1.39	1.31	1.82	1.25	1.10	1.29
BLAST	8	1.0 (1380s)	1.28	1.30	1.27	1.30	1.45	1.52	1.29
	16	1.0 (706s)	1.22	1.22	1.22	1.27	1.98	1.98	1.22
	32	1.0 (365s)	1.26	1.30	1.22	1.13	1.97	1.91	1.20

Figure 4: (Normalized, 1st column) Heatmap showing how different resource controllers (Cgroups 95-05 80-20 and 05-95, random and parallel freezer, and BLB) affect total run times for Montage and BLAST as scale increases (# allocated cores). Each column represents how resource shares (in terms of execution time normalized against the isolated case) were distributed among higher (Montage) priority and lower (BLAST) priority jobs. The last three columns represent specific controllers where application structure is taken into consideration. The larger the numerical value, the longer the application takes to complete.

in isolation. As shown in this figure, the no-control policy gives good performance, but does not preserve application priorities. The static policies (95-5, 80-20, and 5-95) give good results for the prioritized application (Montage), but are inflexible and add overhead to the lower priority jobs (BLAST) or vice versa for 5-95, in particular when more cores are used. The parallel and random freezers protect the performance of the high priority application (Montage) well, but yield very poor performance for the low priority one (BLAST). Our BLB method achieves good performance for Montage, reasonable good for BLAST, and achieves stable results for varying number of cores. Based on these observations, we next study the performance of our proposed method for larger cluster sizes. Henceforth, Slurm (vanilla) corresponds to the isolated case, Slurm (collocated) to the no-control scenario, and Hybrid to our BLB method.

Tables 1 and 2 summarize the results for the generated workload model. In the cluster experiments, Slurm's default queue (Vanilla) is used as a baseline for the other queues, as jobs do not share the node and are submitted and scheduled as done by default in Slurm. In Table 1 the average waiting time is reduced with 42% for our hybrid approach compared to vanilla Slurm, and the average walltime increased by 9%, both due to the resource sharing. In comparison, collocated slurm reduces average waiting time with 52%, but suffers an 162% increase in average walltime that leads to many jobs being killed as the overrun their allocations, which is further explained in Table 2. This table shows the queues makespan, job throughput, CPU utilization, and

Table 1: Summary for five different workloads (a mix of BLAST, Montage, NEK5000, Statistics, and Synthetic)

Application	**Number of Cores**	**Average Waiting Time Slurm (Vanilla)**	**Average Waiting Time Slurm (Collocated)**	**Average Waiting Time Hybrid Architecture**	**Average Walltime Slurm (Vanilla)**	**Average Walltime Slurm (Collocated)**	**Average Walltime Hybrid Architecture**
Montage	[32 - 128]	3569 s	1850 s	2011 s	1331 s	3783 s	1560 s
BLAST	[32 - 512]	3985 s	1075 s	2321 s	2833 s	3683 s	3400 s
NEK5000	[32 - 128]	4235 s	3081 s	3387 s	2877 s	13005 s	3061 s
Statistics	[32 - 128]	2441 s	1092 s	1011 s	1539 s	1711 s	1623 s
Synthetic	[32 - 512]	3382 s	1396 s	1450 s	723 s	955 s	893 s
Average	*250*	*3522 s*	*1699 s*	*2036 s*	*1861 s*	*4627 s*	*2107 s*

Table 2: Average queues makespan, throughput (# jobs/total makespan(s)) and missed deadlines for each allocation approach.

Queue	**Total Workload Makespan (s)**	**Job Throughput (#Jobs/time[s])**	**CPU Utilization (%)**	**Missed Deadlines (%)**
Slurm (Vanilla)	12972	$4.2 * 10^{-4}$	69	**0**
Slurm (Collocated)	21652	$4.5 * 10^{-4}$	**90**	43
Hybrid Architecture	**11380**	$\mathbf{1.2 * 10^{-3}}$	84	**0**

missed deadlines, with the best result(s) for each category highlighted in bold text. Here it can be seen that compared to vanilla Slurm, our approach reduces the makespan with 12% and improves utilization from 69% to 84%, while still meeting all job deadlines. In contrast, the collocated Slurm achieves an impressive resource utilization of 90%, but at the expense of a very long makespan (67% longer than vanilla) and thus poor application performance with 43% of jobs missing deadlines. This illustrates how our approach provides a controlled way to increase datacenter utilization without affecting job performance.

Figure 5 illustrate that that these performance numbers are stable also for different cluster sizes. When we increase the total number of cores in the cluster from 128 to 256 and later to 512, with the number of jobs and/or cores per job increasing accordingly, we observe only minor variations in the makespan and resource utilization for all methods. Most importantly, the observation that our hybrid approach achieves a shorter makespan than vanilla Slurm, with much higher resource utilization, holds for all studied cluster sizes. The cost for guaranteeing the best performance possible can be seen in the average (queue) waiting time for Slurm: one either sacrifices utilization for performance (Vanilla in Table 1), or performance for utilization (Collocated in Table 1). As Slurm does not come with an user-level scheduler for controlling how processes can be scheduled by the OS, datacenter operators often go for a common denominator for what satisfies most users. In contrast, our approach provides a trade-off between utilization and performance, where we actively profile and control resource usage according to the variations in workload.

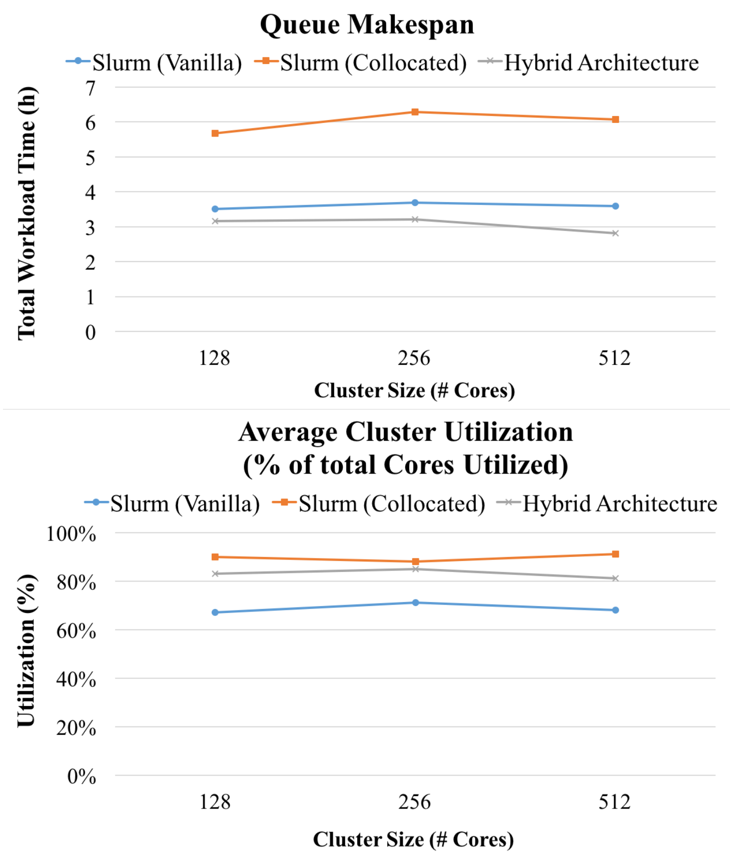


Figure 5: Total (average) queue makespan (left) and average cluster utilization (right) for each cluster size, approach, and workload mix.

5 Discussion

The experimental results highlight the advantages of the proposed hybrid architecture. By constantly adapting to current workload resource utilization (CPU), the proposed approach improves resource utilization over other configurations (Vanilla and Collocated) by at least 15% (from 69% to 84%), with negligible performance overheads in terms of longer wall clock times, and no missed deadlines. Integrating two different resource managers in the same cluster, with different objectives and isolation guarantees is a difficult problem. From an administrator's perspective, the idea of allowing multiple distributed applications independently developed and having their own scheduling policies and requirements to share resources is very challenging. The datacenter operator must enforce isolation among different applications, which can be the limiting factor as HPC clusters usually have more restrictions and are less scalable than cloud clusters and normal distributed and cloud-aware platforms and applications. In addition, the rate of additional failures that a DI scheduler may experience should be studied, as interference may happen more frequently as HPC jobs are more sensitive to disturbances. An additional problem may rise while sampling resource usage metrics: performance degradation. Depending on the frequency of sampling application performance could degrade, though Perf is very lightweight: all of our experiments generated ten's of megabytes of data. To protect the performance of HPC jobs, the amount of data gathered should be reduced by finding an optimal rate of sampling over time, or by controlling it in order to not hurt HPC jobs Service Level Objectives (SLOs) and constraints.

As shown in the evaluation, our approach is most beneficial for scientific workflows. These jobs are composed of sets of different jobs with different patterns of resource requirements and utilization per job submission. However, workflows require domain knowledge and good understanding of hardware optimization to run efficiently in various HPC clusters, since it is the common scenario among research centers. We believe these kinds of jobs are interesting targets for hybrid resource managers in future developments, in particular when it comes to leveraging the characteristics of workflows and adaptive jobs to improve scheduler and cluster utilization. From the application users' point of view, our architecture can be run as normal user. The authors have actually set the architecture up in two large-scale HPC centers, but restrictions to enabling cgroup controls do not allow key architectural features to be evaluated. On this point, once users has access to cgroup namespace control, they can run multiple workflows over the resource she owns, controlling them as needed.This has the advantage of reducing operator burden for supporting a higher number of job queues, as it is the case in the cluster-wide scenario. Although the proposed architecture is being used to improve overall cluster utilization and unlike more robust and industry deployed resource managers such as Kubernetes, these characteristics show that the hybrid architecture can be deployed in the userspace. As such, its advantages and flexibility can also be validated from different user perspectives. Finally, we focused on implementing the architecture components by using a simple and scalable statistical model for the IE. However, as future work we propose to infer more sophisticated probability distributions by monitoring each job performance overtime, which would help with performance variability. This can be important for scientific experiments that rely on predictable hardware performance. Thus, future studies should analyze the impact our solution would have on such (performance wise) sensitive workloads.

6 Conclusions

The fast growing interest on datacenter management from both public and industry together with the rapid expansion in scale and complexity of infrastructure and the services being provided on them have made monitoring, profiling, controlling, and provisioning compute resources dynamically at runtime into a challenging task. As Data Intensive applications resource needs grows, HPC's optimized premises offer promising and powerful capabilities. In this paper we have proposed a way to enable

DI jobs to not only share the same resources with HPC jobs, but to efficiently identify spare resources and throttle their availability. Our work enables running jobs with completely different performance requirements in the same cluster while not disturbing higher priority jobs, in this case HPC jobs that require predictable allocations and are not fault tolerant in general. We believe further studies are required in this area, for instance to understand how long the IE model needs to keep historical data in order to address jobs with highly dynamic changing behaviors. Even though our current BLB model is scalable, another future direction would be to try to run it completely distributed in local nodes of the cluster without a need for a central IE. This way we can detect outliers such as interference faster and with much lower overhead. Additionally, we have chosen some constants in our model, e.g., the confidence. Another area of future work is a more general approach where an estimation of the best confidence that works per job is estimated and used for cluster scheduling.

References

[18] *Montage - An astronomical image mosaic engine*. 2018. URL: http://montage.ipac.caltech.edu/.

[Alt+97] Stephen F Altschul, Thomas L Madden, Alejandro A Schäffer, Jinghui Zhang, Zheng Zhang, Webb Miller, and David J Lipman. "Gapped BLAST and PSI-BLAST: a new generation of protein database search programs". In: *Nucleic acids research* (1997).

[Amv+17] George Amvrosiadis, Jun Woo Park, Gregory R Ganger, Garth A Gibson, Elisabeth Baseman, and Nathan DeBardeleben. "Bigger, Longer, Fewer: what do cluster jobs look like outside Google?" In: (2017).

[AW17] Haripriya Ayyalasomayajula and Karlon West. "Experiences running different work load managers across Cray Platforms". In: *Cray User Group Conference (CUG'17)*. 2017.

[Ber17] Evan Berkowitz. "METAQ: Bundle Supercomputing Tasks". In: *arXiv preprint arXiv:1702.06122* (2017).

[Bha+13] Abhinav Bhatele, Kathryn Mohror, Steven H. Langer, and Katherine E. Isaacs. "There Goes the Neighborhood: Performance Degradation Due to Nearby Jobs". In: *International Conference on High Performance Computing, Networking, Storage and Analysis*. SC '13. Denver, Colorado: ACM, 2013. ISBN: 978-1-4503-2378-9. DOI: 10.1145/2503210.2503247.

[Bre+12] Alex D Breslow, Leo Porter, Ananta Tiwari, Michael Laurenzano, Laura Carrington, Dean M Tullsen, and Allan E Snavely. "The case for colocation of hpc workloads". In: *Concurrency and Computation: Practice and Experience Preprint* (2012).

[BTG13] James M Brandt, Thomas Tucker, and Ann C Gentile. *Lightweight Distributed Metric Service (LDMS): Run-time Resource Utilization Monitoring.* Tech. rep. https://www.osti.gov/servlets/purl/1106397. Sandia National Lab.(SNL-CA), Livermore, CA (United States); Sandia National Lab.(SNL-NM), Albuquerque, NM (United States), 2013.

[Bur+16] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. "Borg, omega, and kubernetes". In: *Queue* 14.1 (2016), p. 10.

[CB01] Walfredo Cirne and Francine Berman. "A model for moldable supercomputer jobs". In: *Proceedings of the 15th International Parallel and Distributed Processing Symposium*. IEEE. 2001.

[Com+16a] Isaıas Comprés, Ao Mo-Hellenbrand, Michael Gerndt, and Hans-Joachim Bungartz. "Infrastructure and API Extensions for Elastic Execution of MPI Applications". In: *23rd European MPI Users' Group Meeting*. EuroMPI 2016. Edinburgh, United Kingdom: ACM, 2016. ISBN: 978-1-4503-4234-6. DOI: 10.1145/2966884.2966917.

[Com+16b] Isaıas Comprés, Ao Mo-Hellenbrand, Michael Gerndt, and Hans-Joachim Bungartz. "Infrastructure and api extensions for elastic execution of mpi applications". In: *23rd European MPI Users' Group Meeting*. ACM. 2016.

[DG08] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: simplified data processing on large clusters". In: *Communications of the ACM* (2008).

[DK13] Christina Delimitrou and Christos Kozyrakis. "Paragon: QoS-aware scheduling for heterogeneous datacenters". In: *ACM SIGARCH Computer Architecture News* (2013).

[DK14] Christina Delimitrou and Christos Kozyrakis. "Quasar: resource-efficient and QoS-aware cluster management". In: *ACM SIGARCH Computer Architecture News* (2014). ISSN: 0163-5964. DOI: 10.1145/2654822.2541941.

[Eme+13] Joseph Emeras, Cristian Ruiz, Jean-Marc Vincent, and Olivier Richard. "Analysis of the jobs resource utilization on a production system". In: *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer. 2013.

[Eva+14] Todd Evans, William L Barth, James C Browne, Robert L DeLeon, Thomas R Furlani, Steven M Gallo, Matthew D Jones, and Abani K Patra. "Comprehensive resource use monitoring for hpc systems with tacc stats". In: *Proceedings of the First International Workshop on HPC User Support Tools*. IEEE Press. 2014, pp. 13–21.

[Fis+08] P Fischer, J Kruse, J Mullen, H Tufo, J Lottes, and S Kerkemeier. "Nek5000: Open source spectral element CFD solver". In: *Argonne National Laboratory, Mathematics and Computer Science Division, Argonne, IL, see https://nek5000.mcs.anl.gov* (2008).

[FR96] Dror G Feitelson and Larry Rudolph. “Toward convergence in job schedulers for parallel supercomputers”. In: *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer. 1996.

[FTK14] Dror G Feitelson, Dan Tsafrir, and David Krakov. “Experience with using the parallel workloads archive”. In: *Journal of Parallel and Distributed Computing* 74.10 (2014), pp. 2967–2982.

[Gho+11] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. “Dominant Resource Fairness: Fair Allocation of Multiple Resource Types.” In: *Nsdi*. Vol. 11. 2011. 2011, pp. 24–24.

[Her+16] Andrew Herdrich, Edwin Verplanke, Priya Autee, Ramesh Illikkal, Chris Gianos, Ronak Singhal, and Ravi Iyer. “Cache QoS: From concept to reality in the Intel® Xeon® processor E5-2600 v3 product family”. In: *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE. 2016, pp. 657–668.

[Hin+09] Benjamin Hindman, Andrew Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Scott Shenker, and Ion Stoica. “Nexus: A common substrate for cluster computing”. In: *Workshop on Hot Topics in Cloud Computing*. 2009.

[Hin+11] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy H Katz, Scott Shenker, and Ion Stoica. “Mesos: A platform for fine-grained resource sharing in the data center.” In: *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 2011.

[Hin18] Jonathan Hines. “Stepping up to Summit”. In: *Computing in Science & Engineering* 20.2 (2018), pp. 78–82.

[Jha+14] Somesh Jha, Jian Qiu, Andre Luckow, Pradeep Mantha, and Geoffrey C Fox. “A tale of two data-intensive paradigms: Applications, abstractions, and architectures”. In: *IEEE BigData Congress, 2014*. 2014.

[KJ11] J Zico Kolter and Matthew J Johnson. “REDD: A public data set for energy disaggregation research”. In: *Workshop on Data Mining Applications in Sustainability (SIGKDD), San Diego, CA*. Citeseer. 2011.

[Kle+12] Ariel Kleiner, Ameet Talwalkar, Purnamrita Sarkar, and Michael Jordan. “The big data bootstrap”. In: *arXiv preprint arXiv:1206.6415* (2012).

[Mar+11] Jason Mars, Lingjia Tang, Robert Hundt, Kevin Skadron, and Mary Lou Soffa. “Bubble-Up: Increasing Utilization in Modern Warehouse Scale Computers via Sensible Co-locations”. In: *44th Annual IEEE/ACM International Symposium on Microarchitecture - MICRO-44 ’11* (2011), p. 248. ISSN: 10724451. DOI: 10.1145/2155620.2155650.

[Mel09] Arnaldo Carvalho de Melo. “Performance counters on Linux”. In: *Linux Plumbers Conference*. Vol. 118. 2009.

[Men07] Paul B Menage. “Adding generic process containers to the linux kernel”. In: *Linux Symposium*. Vol. 2. Citeseer. 2007, pp. 45–57.

[Mer+17] Michael Mercier, David Glesser, Yiannis Georgiou, and Olivier Richard. “Big Data and HPC collocation: Using HPC idle resources for Big Data Analytics”. In: *IEEE BigData 2017*. 2017.

[NK10] Ripal Nathuji and Aman Kansal. “Q-Clouds : Managing Performance Interference Effects for QoS-Aware Clouds”. In: *5th European conference on Computer systems* (2010), pp. 237–250. DOI: 10.1145/1755913.1755938.

[Ous+13] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. “Sparrow: distributed, low latency scheduling”. In: *Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM. 2013, pp. 69–84.

[Pal+15] Jeffrey T Palmer, Steven M Gallo, Thomas R Furlani, Matthew D Jones, Robert L DeLeon, Joseph P White, Nikolay Simakov, Abani K Patra, Jeanette Sperhac, Thomas Yearke, et al. “Open XDMoD: A tool for the comprehensive management of high-performance computing resources”. In: *Computing in Science & Engineering* (2015).

[Pat84] Anthony T Patera. “A spectral element method for fluid dynamics: laminar flow in a channel expansion”. In: *Journal of computational Physics* (1984).

[PPS15] David C Parkes, Ariel D Procaccia, and Nisarg Shah. “Beyond dominant resource fairness: Extensions, limitations, and indivisibilities”. In: *ACM Transactions on Economics and Computation (TEAC)* 3.1 (2015), p. 3.

[Pra+15] Suraj Prabhakaran, Marcel Neumann, Sebastian Rinke, Felix Wolf, Abhishek Gupta, and Laxmikant V Kale. “A batch system with efficient adaptive scheduling for malleable and evolving applications”. In: *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE. 2015.

[Reu+16] Albert Reuther, Chansup Byun, William Arcand, David Bestor, Bill Bergeron, Matthew Hubbell, Michael Jones, Peter Michaleas, Andrew Prout, Antonio Rosa, et al. “Scheduler technologies in support of high performance data analysis”. In: *High Performance Extreme Computing Conference (HPEC), 2016 IEEE*. IEEE. 2016.

[Reu+18] Albert Reuther, Chansup Byun, William Arcand, David Bestor, Bill Bergeron, Matthew Hubbell, Michael Jones, Peter Michaleas, Andrew Prout, Antonio Rosa, et al. “Scalable system scheduling for HPC and big data”. In: *Journal of Parallel and Distributed Computing* 111 (2018), pp. 76–92.

[RS00] Larry Rudolph and Paul H Smith. “Valuation of ultra-scale computing systems”. In: *JSSPP*. Springer. 2000, pp. 39–55.

[Sch+13] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. “Omega: flexible, scalable schedulers for large compute clusters”. In: *8th ACM European Conference on Computer Systems*. ACM. 2013, pp. 351–364.

[Sha+13] Aamer Shah, Felix Wolf, Sergey Zhumatiy, and Vladimir Voevodin. “Capturing inter-application interference on clusters”. In: *Cluster Computing (CLUSTER), 2013 IEEE International Conference on*. IEEE. 2013, pp. 1–5.

[Sta06] Garrick Staples. “TORQUE Resource Manager”. In: *2006 ACM/IEEE Conference on Supercomputing*. SC ’06. Tampa, Florida: ACM, 2006. ISBN: 0-7695-2700-0. DOI: 10.1145/1188455.1188464.

[Tiw+13] A Tiwari, M Schulz, L Carrington, L Tang, and J Mars. *Enabling Fair Pricing on HPC Systems with Node Sharing*. Tech. rep. Lawrence Livermore National Laboratory (LLNL), Livermore, CA, 2013.

[Val90] Leslie G Valiant. “A bridging model for parallel computation”. In: *Communications of the ACM* (1990).

[Vav+13] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, et al. “Apache Hadoop Yarn: Yet another resource negotiator”. In: *4th annual Symposium on Cloud Computing*. ACM. 2013, p. 5.

[Wan+10] Chao Wang, Frank Mueller, Christian Engelmann, and Stephen L Scott. “Hybrid checkpointing for MPI jobs in HPC environments”. In: *IEEE 16th International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE. 2010.

[Wea13] Vincent M Weaver. “Linux perf_event features and overhead”. In: *The 2nd International Workshop on Performance Analysis of Workload Optimized Systems, FastPath*. Vol. 13. 2013.

[Whi12] Tom White. *Hadoop: The definitive guide*. ”O’Reilly Media, Inc.”, 2012.

[Yan+13] Hailong Yang, Alex Breslow, Jason Mars, and Lingjia Tang. “Bubble-flux: Precise online qos management for increased utilization in warehouse scale computers”. In: *ACM SIGARCH Computer Architecture News*. Vol. 41. 3. ACM. 2013, pp. 607–618.

[YJG03] Andy B Yoo, Morris A Jette, and Mark Grondona. “Slurm: Simple linux utility for resource management”. In: *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer. 2003, pp. 44–60.

[Zah+10] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. “Spark: Cluster computing with working sets.” In: *HotCloud* (2010).

[Zha+13] Xiao Zhang, Eric Tune, Robert Hagmann, Rohit Jnagal, Vrigo Gokhale, and John Wilkes. “CPI 2: CPU performance isolation for shared compute clusters”. In: *8th ACM European Conference on Computer Systems*. ACM. 2013.