Hybrid Workload Scheduling on HPC Systems

Yuping Fan, Zhiling Lan
Illinois Institute of Technology
Chicago, IL
yfan22@hawk.iit.edu, lan@iit.edu

Paul Rich, William Allcock
Argonne National Laboratory
Lemont, IL
{richp,allcock}@anl.gov

Michael E. Papka
Argonne National Laboratory
Northern Illinois University
papka@anl.gov

***Abstract*—Traditionally, on-demand, rigid, and malleable applications have been scheduled and executed on separate systems. The ever-growing workload demands and rapidly developing HPC infrastructure trigger the interest of converging these applications on a single HPC system. Although allocating the hybrid workloads within one system could potentially improve system efficiency, it is difficult to balance the tradeoff between the responsiveness of on-demand requests, incentive for malleable jobs, and the performance of rigid applications. In this study, we present several scheduling mechanisms to address the issues involved in co-scheduling on-demand, rigid, and malleable jobs on a single HPC system. We extensively evaluate and compare their performance under various configurations and workloads. Our experimental results show that our proposed mechanisms are capable of serving on-demand workloads with minimal delay, offering incentives for declaring malleability, and improving system performance.**

***Index Terms*—cluster scheduling, high-performance computing, on-demand jobs, rigid jobs, malleable jobs**

I. Introduction

The tremendous compute power with high bandwidth memory and enormous storage capabilities makes high performance computing (HPC) facilities ideal infrastructures for various types of applications. The main tenant of HPC systems is *batch applications*, which are tightly coupled parallel jobs and are rigid in size. *On-demand applications* are time-critical applications requiring quick response and thus are used to running on their dedicated clusters. As the sizes of on-demand applications are rapidly expanding in recent years, the dedicated clusters cannot keep up with the rapid expansion in on-demand applications. As a result, HPC system becomes a more practical solution for on-demand applications. *Malleable applications* are loosely coupled applications consisting of a series of tasks and therefore they can adapt their sizes to changes in hardware availability. Malleable applications are typically running in datacenters. In recent years, an increasing number of HPC systems are equipped with accelerators. The superior computing power combined with the emerging accelerators makes HPC systems an attractive alternative for malleable applications [1], [2].

The production HPC cluster schedulers, such as Slurm, Moab/TORQUE, PBS, and Cobalt [3]–[6], adopt the traditional batch job scheduling model, where users request a fixed amount of resources for a specific amount of time, while the scheduler decides when and where to run each job based on job priority and system availability. A number of studies attempt to address the hybrid workload scheduling on a single HPC system. Research on co-scheduling rigid and on-demand applications often aims at the high responsiveness of on-demand jobs. The common strategies include predicting on-demand jobs' requests, reserving resources for on-demand jobs, and preempting rigid jobs to make room for on-demand jobs [7]–[11]. Other studies focus on co-scheduling malleable jobs with rigid jobs on HPC systems [12]–[23]. Unfortunately, these studies do not address the problem of co-scheduling all three types of applications, i.e., on-demand jobs, rigid jobs, and malleable jobs. Hence, the scheduling implications of co-running these applications are unknown.

Cluster scheduling consists of two components: job scheduler and resource manager, where job scheduler determines when and which user jobs should be allocated to system resources, and resource manager monitors and manages resource allocations. Executing hybrid workloads on a single HPC system has several benefits, such as supporting ever-increasing on-demand job sizes, reducing resource fragmentation, and improving system utilization. However, this is a challenging task at both job scheduling level and resource management level. The job scheduler needs to maintain the delicate balance between several conflicting objectives, i.e., quick response to on-demand jobs, high system utilization, the incentive for shrinking malleable jobs, and low impact on rigid jobs. The resource manager has to execute the more complicated and frequent operations from job scheduler, i.e., start, preemption, shrink, and expansion of user jobs.

In this paper, we concentrate on addressing HPC hybrid workloads problem from the job scheduling aspect. We present six mechanisms for *co-scheduling all three types of applications (i.e, batch, on-demand, and malleable applications)*. Our design intends to meet the demands from different applications, while also maximizing system utilization. Our proposed mechanisms are designed to be used in conjunction with the existing site policy: while a site policy determines the order of waiting jobs, our mechanisms manipulate running jobs in order to provide timely service to on-demand jobs with minimal negative impact on rigid and malleable jobs. Our design is based on the fact that it is often possible for on-demand jobs to determine their demand within a short time (15-30 minutes) before their actual arrivals [7]. Upon receiving on-demand job's advance notice, we provide both non-invasive and invasive mechanisms to reserve resources for the on-demand job. Once an on-demand job arrives, we provide several mechanisms to immediately vacate nodes

from running malleable and rigid jobs. By combining the mechanisms used at on-demand job's advance notice and its arrival, we propose six mechanisms to handle hybrid workload scheduling problems.

Moreover, we conduct a series of trace-based simulations using various workloads generated based on real workload traces collected from Theta [24] at Argonne Leadership Computing Facility (ALCF) and Cori [25] at National Energy Research Scientific Computing (NERSC) . These experiments not only allow us to extensively evaluate different co-scheduling methods, but also help us gain valuable insights regarding co-scheduling different workloads on HPC systems. The results show that all of the proposed mechanisms achieve quick responsiveness for on-demand jobs. Additionally, the results reveal the impact of different mechanisms on system performance and the performance on malleable and rigid jobs. More importantly, we provide valuable insights for choosing these mechanisms under different situations.

II. Related Work and Challenges

A. HPC Application Types

Rigid job is the most common type of job in HPC environments [26]. Rigid jobs have fixed resource requirements throughout their life cycle. Most parallel applications, such as extreme-scale scientific simulations and modeling, are rigid in nature, requiring inter-process communication through message passing, and checkpointing for fault tolerance. They are tightly coupled applications that cannot be decomposed to a series of small-sized tasks and are prone to failure due to their sizes. In order to handle hardware failures, rigid applications checkpoint regularly and restart from the latest checkpoint in the event of an interruption.

On-demand job is a time-critical application needed to be completed in the shortest time possible. An example of the on-demand jobs is data analytical workloads after experiments [7]. Traditionally, to ensure high responsiveness, on-demand jobs are running on dedicated small clusters. This leads to very low cluster utilization. The rapid experimental expansion requires increasingly large computing capabilities, which cannot be fulfilled by small clusters. The use of large-scale HPC systems becomes a viable solution for the ever-increasing on-demand workloads.

Malleable job is another type of parallel job whose sizes can adapt to the number of nodes assigned to them. A malleable job specifies the minimum and the maximum number of nodes. They can shrink down to the minimum sizes or expand up to the maximum sizes based on resource availability. Typically, a malleable job consists of loosely coupled small-sized tasks and the running tasks can be dynamically adjusted based on the assigned nodes. In addition, preemption of malleable jobs causes less overhead than rigid jobs, because they can skip over the finished tasks and resume from the interrupted tasks. The typical examples of malleable jobs are high throughput jobs [27], multi-task workflows [14], machine learning applications, and hyperparameter searches in deep neural networks. While traditionally separated infrastructures have been used for rigid jobs and malleable jobs [28], [29], the next-generation HPC systems provide not only tremendous compute power on a single node (CPU and GPU), but also enormous high bandwidth memory, making them efficient platforms for both types of workloads. As a result, malleable applications are gaining increasing traction on HPC systems in recent years.

B. Job Scheduling in HPC

HPC job scheduling is traditionally designed to manage and assign rigid jobs to resources. The resource allocation is commonly at the granularity of a node. Well-known HPC schedulers include Slurm, Moab/TORQUE, PBS, and Cobalt [3]–[6]. When submitting a job, a user is required to provide job size and job runtime estimate. At each scheduling instance, the scheduler orders the jobs in the queue according to site policies and resource availability and executes jobs from the head of the queue [30]. The most widely used HPC job scheduling policy is First Come First Serve (FCFS) with EASY backfilling [31]. FCFS sorts the jobs in the queue according to their arrival times, while backfilling is often used in conjunction with reservation to enhance system utilization. Backfilling allows subsequent jobs in the queue to move ahead under the condition that they do not delay the existing reservations.

In the realm of executing on-demand jobs and rigid jobs on HPC systems, several groups have proposed to statically or dynamically reserve resources for on-demand requests. Dynamical reservation was achieved by predicting the on-demand request patterns [7], [8]. Another widely adopted technique to ensure timeliness of on-demand jobs is to preempt rigid jobs [9]–[11]. In terms of accommodating malleable jobs and rigid jobs on HPC systems, several attempts have been made to shrink malleable jobs in order to reduce resource fragmentation problems [12], [15], [21], [22]. To the best of our knowledge, this is the first attempt to co-schedule all three types of jobs (i.e., rigid, on-demand, and malleable) on a single HPC system.

Our work also differs from existing cloud resource managers like Mesos and Kubernetes [32], [33]. Cloud resource managers commonly allow jobs to share nodes. As a result, solutions for addressing bursty on-demand requests often rely on co-scheduling mixed workloads on a single node via containers or virtual machines. This is very different from HPC where a bare metal mode with exclusive node access is used for running jobs.

Finally, some studies proposed cross-platform solutions. For example, Ambati et al. optimized operating costs and reduced wait time by pushing some workloads to cloud providers when a HPC system was too busy to handle the bursty requests [34]. However, these solutions are difficult to apply to HPC workloads, especially rigid jobs, which are highly optimized based on specific systems and configurations. Additionally, our study focused on extracting the best performance from a fixed amount of resources rather than seeking additional resources.

C. Technical Challenges

Managing hybrid workloads on a single large-scale HPC system offers several benefits, such as boosting system utilization, mitigating system fragmentation, and reducing job turnaround time. However, the hybrid workloads also pose new challenges.

- *Maximize instant start ratio of on-demand jobs.* One of the primary goals is to maximize the number of on-demand jobs that can start instantly upon their arrival. By moving the dedicated allocation of on-demand requests to a common resource pool, other types of jobs can utilize these resources and improve system utilization. However, the high system utilization also makes it extremely difficult to achieve high on-demand instant start ratio.
- *Minimize resource waste.* To accommodate the hybrid workloads, a proper mechanism must take advantage of job shrink, expansion, and checkpointing strategies. These strategies come with overheads. For example, to make room for time-critical on-demand requests, we could preempt running rigid/malleable jobs and resume them from the latest checkpoints. These preempted jobs will lose the computation after the checkpoints. Hence, an effective solution must take the resource waste into consideration when choosing running jobs for preemption.
- *Incentive of being malleable.* For those jobs that are capable of being adjusted to different sizes, users can either declare them as rigid jobs or malleable jobs. The designed strategies need to provide incentives for users to declare them as malleable jobs by guaranteeing better job performance, e.g., lower average job turnaround time. This could discourage users from lying about their job types.
- *Quick decision making.* To fulfill time-critical on-demand requests, the scheduler has to rapidly choose running jobs to make room for on-demand jobs. Malleable jobs can either be preempted or shrunk, which leads to additional complexity and makes the problem non-trivial. A proper design must be scalable and be capable of making high-quality decisions in a short time (e.g., in seconds).

III. Methodology

In this section, we first formally define our hybrid workload scheduling problem in §III-A. We then present the six scheduling mechanisms to solve this problem in §III-B.

A. Problem Formulation

Suppose an HPC system has N identical nodes. Independent jobs $J_1, J_2, ..., J_n$ arrive and are scheduled in order. We assume that jobs cannot share nodes and thus jobs must be allocated an integral number of nodes. Jobs can be classified into three categories:

- **Rigid job**: When submitting a rigid job, a user is required to provide two pieces of information: the number of nodes n and job runtime estimate $t_{estimate}$. A rigid job requires a fixed number of nodes, which cannot be adjusted during execution. Job's actual runtime t_{actual} cannot exceed the job runtime estimate ($t_{actual} \leq t_{estimate}$); otherwise, the job will be killed when reaching the runtime estimate [35]. At the beginning of job execution, a job needs some time t_{setup} to set up communication and coordination. During job execution, the job might take regular checkpoints at frequency t_f. In case of interruption, the resumed job will first set up communication in t_{setup} time and then resume from the latest checkpoint. As a result, the resumed job will lose the computation between the latest checkpoint and the preempted time.
- **On-demand job**: On-demand jobs are time-critical applications, which needed to start within a very short time after submission. On-demand jobs are often possible to determine their resource need within a short time (15-30 minutes) before submission. Advance notice includes the following information: the estimated job arrival time, job size, and job runtime estimate. Based on the on-demand job's estimated arrival time and actual arrival time, on-demand jobs can be categorized into four groups as shown in Figure 1, i.e., without advance notice, with accurate advance notice, arrive early, and arrive late.

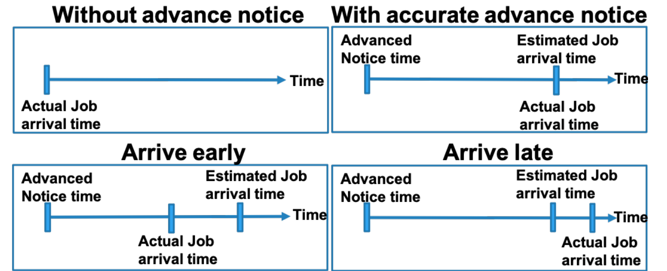

Fig. 1: Four types of on-demand jobs.

- **Malleable job**: When submitting a malleable job, a user provides the following information: minimum job size n_{min}, maximum job size n_{max}, job estimate runtime when running at maximum job size $t_{estimate}$. Similar to rigid jobs, we consider setup time at the beginning of the execution. A malleable job is able to run on any integer nodes between minimum job size and maximum job size ($n_{min} \leq n \leq n_{max}$). We assume the linear speedup in addition to the constant setup overhead. Therefore, we can model job's actual runtime as: $t_{actual} = t_{single}/n + t_{setup}$. Here, t_{single} is the application's runtime on a single compute node. Note that the size of malleable jobs can be adjusted before or during execution according to scheduling policies, which is slightly different from the well-adopted definition of malleable jobs in [26]. A malleable job typically consists of small-sized tasks and the overhead of changing job size is negligible, and thus it is reasonable to assume no overhead involves in job expansion or shrink. In case of preemption, We adopt Amazon's two minutes warning strategy on spot instance [36]. The scheduler provides two minutes for malleable jobs to make a checkpoint. The resumed malleable jobs will first take t_{setup} to set up and then will resume from the previous preempted time. Note that we take the different checkpointing strategies for rigid jobs and malleable jobs. This is because the checkpointing overhead of malleable

jobs is, in general, much lower and more predictable than that of rigid jobs. Two minutes is sufficient for a malleable job to store its states to disk and thus it can avoid regular checkpointing overheads.

The scheduling problem we study is to allocate resources to on-demand jobs as soon as possible by reserving available nodes and preempting or shrinking running jobs. We aim to respond to on-demand jobs in a timely manner while minimizing the negative impact on rigid and malleable jobs.

B. Mechanisms

We design our hybrid workloads scheduling problem as a series of decisions triggered by *four types of events of on-demand jobs: advance notice, actual arrival, estimated arrival, and completion*. We propose different strategies to handle these events accordingly.

1) Advance notice: The advance notice allows the scheduler to prepare resources for on-demand jobs before their actual arrival. We propose three mechanisms to handle on-demand jobs' advance notice:

- **Do nothing (N)**. This is the baseline strategy. The scheduler ignores advance notice and will handle on-demand jobs later when they actually arrive.
- **Collect-until-actual-arrival (CUA)**. When receiving an on-demand job's advance notice, the scheduler first collects the currently available nodes for this on-demand job. If more nodes are needed, the scheduler will collect nodes released by finished jobs until the requested number of nodes is fulfilled or the on-demand job actually arrives. In case of competition from multiple on-demand jobs, the released nodes will be assigned to the on-demand job with the earliest advance notice.
- **Collect-until-predicted-arrival (CUP)**. Like collect-until-actual-arrival method, this method first reserves the currently available nodes for the on-demand job. If the on-demand job needs more nodes, this method will try to prepare sufficient nodes at its predicted arrival time. First, it will collect the nodes that are expected to be released before the on-demand predicted arrival time. Second, it will preempt running jobs before the on-demand job's predicted arrival time. To preempt which running jobs is determined by preemption overheads. To minimize the preemption overhead, we try to preempt rigid jobs immediately after they make a checkpoint. If the on-demand job arrives earlier than its predicted arrival time, we stop the preparation and use the strategies in the following subsection to collect more nodes.

Figure 2 uses an example to illustrate the differences between CUA and CUP. To improve the system utilization, the nodes reserved for on-demand jobs can be used to run waiting jobs. First, we try to backfill the waiting jobs that are expected to finish before the on-demand job's estimated arrival time. If some reserved nodes are still idle, malleable jobs will be selected to run due to their low preemption overhead and fast draining process. But once the on-demand job arrives, all these jobs have to be preempted immediately.

2) On-demand job's actual arrival: When an on-demand job arrives, the scheduler first checks if there are sufficient available nodes and reserved nodes to run this job. If that is the case, the on-demand job can launch immediately. Otherwise, we propose two strategies to find more nodes for the on-demand job:

- **Preempt-at-actual-arrival (PAA)**: This method lists all currently running malleable and rigid jobs in ascending order of their preemption overheads. For jobs with checkpoints, preemption overhead includes re-computation cost between the preemption and the latest checkpoint and the setup cost. For jobs without checkpoints, preemption overhead is the elapsed time from the job start time to the preemption time. If the total number of the preemptable nodes is not sufficient, we cannot start the on-demand job instantly and have to put it to the front of the queue waiting for additional available nodes. If the preemptable nodes are sufficient, we preempt jobs from the front of the running list until the on-demand request is satisfied. We update the preempted jobs' estimated runtime, keep their original submit time, and automatically resubmit these jobs to the wait queue. The priority of the preempted jobs is determined by the scheduling policy. For example, FCFS might move the preempted jobs to the front of the queue, because they have early first submission times.

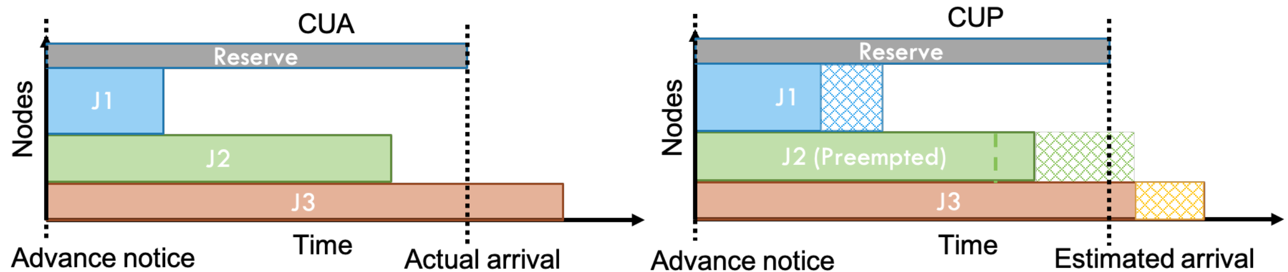

Fig. 2: CUA versus CUP. The solid rectangle is the time actually used by a job; the grid rectangle shows the time between job's actual and estimated finish time. CUA finds running jobs that actually finished before the on-demand job arrival. In this example, the nodes released from J1 and J2 will be reserved. CUP finds running jobs that are estimated to be finished before the on-demand job's estimated arrival time. Hence, CUP first selects J1; since J2 is expected to finish later than the on-demand job's estimated arrival time, it will be preempted immediately after checkpointing (the green dashed line). J2's unfinished computation will be resubmitted and resumed later.

- **Shrink-preempt-at-actual-arrival (SPAA)**: This method first finds all currently running malleable jobs and computes the maximum number of nodes they can supply by shrinking to their minimum sizes. If the supply can meet the on-demand job's request, the running malleable jobs will shrink their sizes as evenly as possible. Once the shrink sizes are determined, the scheduler will linearly adjust the estimated runtime of the shrunk jobs. If the supply cannot meet, we use PAA method to handle the on-demand request.

3) On-demand job's estimated arrival: An on-demand job may arrive late or even do not show up. To preempt deadlocks, if an on-demand job has not arrived after a certain period of time of its estimated arrival time, the scheduler will release the reserved nodes.

4) Completion of on-demand job: For job fairness, once an on-demand job is completed, the on-demand job will try to return its nodes to the lenders. If a job was preempted by this on-demand job and is still waiting in the queue, the leased nodes will return to this job and this job will resume immediately if possible. If a job was shrunk and is still running, we will expand this job to its original size.

By combining three advance notice strategies with two job arrival strategies, we obtain six mechanisms to schedule hybrid workloads on a single HPC system: *N&PAA, N&SPAA, CUA&PAA, CUA&SPAA, CUP&PAA, CUP&SPAA*. Current HPC systems typically require a scheduler to respond in 10-30 seconds [30]. In our experiments, all six mechanisms take less than 10 milliseconds to make a decision and thus are feasible for practical deployment.

IV. Experimental Setup

In this section, we first describe the real workload traces collected from Theta and Cori, and how to generate traces from the real traces to represent various scenarios (§IV-A). We then introduce the baseline configuration for our experiments (§IV-B) and our simulation environment (§IV-C). Finally, we list the system- and user-centric metrics for evaluation (§IV-D).

A. Workloads

In our study, two real workload traces are used. Table I summarizes the traces collected from two production systems, and Figure 3 gives an overview of job size distributions on these supercomputers. We select these traces as they represent different workload profiles: (1) capability computing focusing on solving large-sized problems, (2) capacity computing solving a mix of small-sized and large-sized problems. The first workload is a two-year job log from Theta [24], a capability computing system located at ALCF. The smallest job allowed on Theta is 128-node [37]. The second trace is a four-month job log from Cori [25], a capacity computing system deployed at NERSC. A majority of its jobs consume one or several nodes (Figure 3).

Since the traces do not include job type information, we generate a series of workloads based on the real traces to cover various job distributions. Studies have been shown that users tend to submit a bunch of on-demand jobs in a short period of time [7]. In order to mimic the bursty on-demand job submission pattern, we group jobs by their project names and assume that all jobs belonging to one project have the same job type. We randomly assign that 10% of projects submit on-demand jobs, 60% of projects submit rigid jobs and the rest of projects submit malleable jobs. We made this assumption because rigid jobs are the main tenant and malleable jobs are emerging in HPC systems. HPC systems can support limited amounts of on-demand requests to ensure their quick response. Figure 4 presents the job type statistics of the randomly generated traces used in our experiments.

TABLE I: Theta and Cori workloads.

	Theta	Cori
Location	ALCF	NERSC
Scheduler	Cobalt	Slurm
System Types	Capability computing	Capacity computing
Compute Nodes	4,392	12,076
Trace Period	Jan. - Dec. 2019	Apr. - Jul. 2018
Number of Jobs	37,298	2,607,054
Max Job Length	1 day	7 days
Min Job Size	128 nodes	1 node

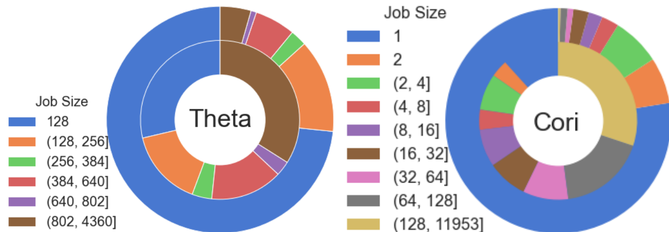

Fig. 3: Job characterization of Theta at ALCF and Cori at NERSC. The outer circle shows the number of jobs in each job size category. The inner circle presents the total core hours consumed by each job size category.

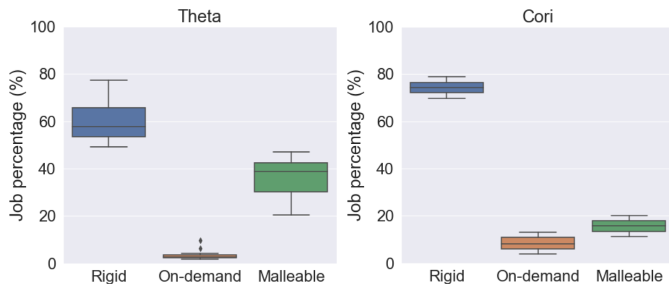

Fig. 4: Job type statistics of the traces used in the experiments.

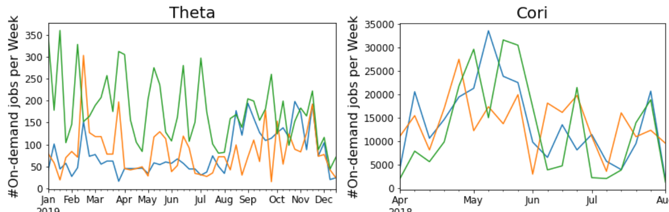

Fig. 5: The number of on-demand jobs submitted per week of three randomly generated traces (denoted by blue, orange, and green lines respectively) on Theta and Cori.

We observe that the job distributions differ significantly on different traces because different projects have significant differences in the number of jobs. Cori log contains 76% of single-node jobs, which cannot change their job sizes. For those single-node jobs be originally assigned as malleable jobs, we randomly re-assign them as either rigid or on-demand jobs. As a result, Cori has a higher percentage of rigid jobs and a lower percentage of malleable jobs compared to Theta. Additionally, Cori's malleable jobs are, on average, larger in size than rigid and malleable jobs. Figure 5 shows the number of on-demand jobs submitted per week of three randomly generated traces. The submission patterns of the different traces vary significantly and all traces show the bursty on-demand job submission pattern. This enables us to extensively evaluate our mechanisms under various scenarios.

B. Configuration

In this section, we present the default configurations for different types of jobs. By default, jobs are scheduled by FCFS with EASY backfilling without hardware or software failures. In terms of rigid jobs, their setup overhead is assigned to be 5%-10% of their runtimes. We assume rigid jobs make regular checkpoints at the optimum frequency defined by Daly [38]. Based on our experience and the current literature [39]–[41], we set each checkpointing overhead to 600 seconds if job size is less than 1K nodes; otherwise, we set it to 1200 seconds. If the optimum checkpointing interval is longer than the job runtime, we assume that the job does not make checkpoints.

In terms of on-demand jobs, we equally distribute them into the following categories: without advance notice, with accurate advance notice, arrive early, and arrive late. If an on-demand job arrives early, its arrival time is a random number between its advance notice and estimated arrival time. If an on-demand job arrives late, its arrival time is a random number within 30 minutes after its estimated arrival time. We set the threshold to release the reserved nodes to 10 minutes after the on-demand job's estimated arrival time.

In terms of malleable jobs, we set their maximum job size to be their original requested job size and their minimum job size to be 20% of their maximum size. The setup overhead is a random number between 0%-5% of their runtimes. Note that these configurations are based on our discussion with experienced system managers and administrators at ALCF.

C. Trace-based Simulation

We compare different scheduling mechanisms through trace-based simulation. Specifically, a trace-based, event-driven scheduling simulator called CQSim is used in our experiments [42]. CQSim contains a queue manager and a scheduler that can plug in different scheduling policies. It emulates the actual scheduling environment. A real system takes jobs from user submission, while CQSim takes jobs by reading the job arrival information in the trace. Rather than executing jobs on system, CQSim simulates the execution by advancing the simulation clock according to the job runtime information in the trace.

D. Evaluation Metrics

We evaluate the performance of different mechanisms using several user-level and system-level metrics.

1) **Job turnaround time** is a user-level metric. It measures the interval between job submission and completion time.
2) **On-demand jobs' instant start ratio** is a user-level metric, which is calculated as the ratio between the number of on-demand jobs started instantly and the total number of on-demand jobs. Note that we do not consider the time to preempt/shrink nodes upon the arrival of on-demand jobs.
3) **Preemption ratio** is a user-level metric to measure the percentage of rigid or malleable jobs being preempted.
4) **System utilization** is a system-level metric that measures the ratio of node-hours used for useful job execution to the total elapsed node-hours. Note that system utilization excludes wasted computation due to preemption.

V. EVALUATION

To comprehensively evaluate the six proposed mechanisms, we conduct a series of experiments to compare their performance under various situations/configurations, including different advance notice settings (§V-B), different checkpoint settings (§V-C), different malleable job size range settings (§V-D), and different setup overhead settings (§V-E).

A. Overall Performance

We compare the performance of the five workloads shown in Table II under different on-demand request accuracies in Figure 6 (Theta) and Figure 7 (Cori). In this subsection, we make several interesting observations on overall performance. In the next subsection, we will analyze the impact of advance notice accuracies using these figures.

TABLE II: Distribution of on-demand jobs of different workloads. Take W1 as an example: 70% of on-demand jobs arrive without advance notice; 10% of on-demand jobs arrive with accurate advance notice; 10% of on-demand jobs arrive early; the rest 10% of on-demand jobs arrive late.

	No Notice	Accurate Notice	Arrive Early	Arrive Late
W1	70%	10%	10%	10%
W2	10%	70%	10%	10%
W3	10%	10%	70%	10%
W4	10%	10%	10%	70%
W5	25%	25%	25%	25%

Observation 1. *Compared with FCFS/EASY, the proposed methods boost system utilization and on-demand jobs' instant start ratio, while slightly increasing average job turnaround time.*

The proposed methods improve system utilization by more than 5% on both systems. The on-demand jobs' instant start ratio dramatically increases from 22% to 98% on Theta and from 19% to 99% on Cori. The average jobs turnaround time slightly increases from 15 to 22 hours on Theta and from 2 to 2.5 hours on Cori, due to job preemption and shrink.

Observation 2. *N&PAA has the worst overall performance.*

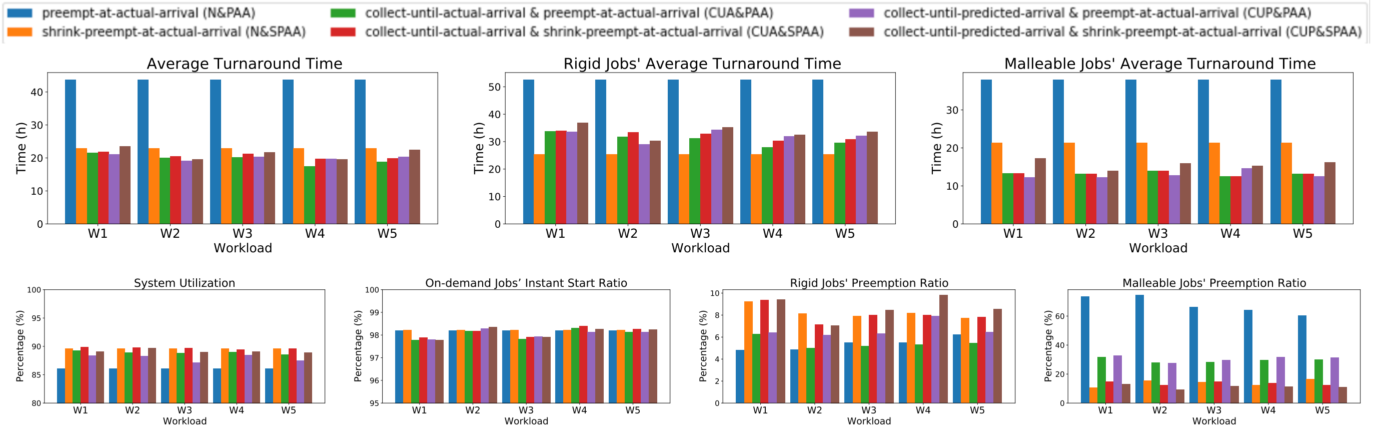

Fig. 6: Scheduling performance on Theta under different advance notice accuracies (shown in Table II). To show the performance under various situations, we repeat the experiment on ten randomly generated traces and the results in this section are averaged.

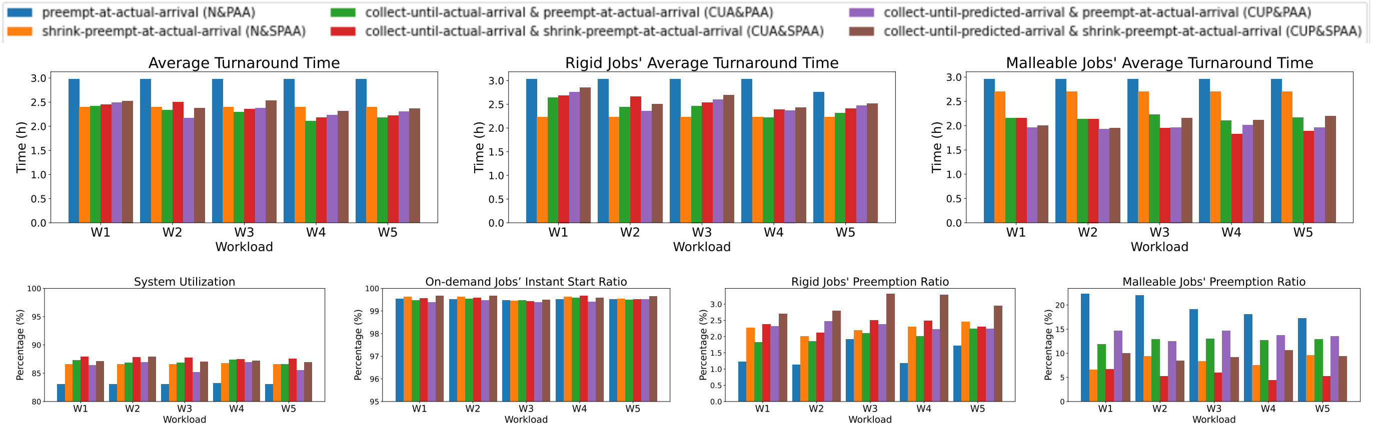

Fig. 7: Scheduling performance on Cori under different advance notice accuracies (shown in Table II).

TABLE III: Baseline performance. Baseline algorithm is FCFS/EASY backfilling without special treatments on on-demand, rigid, and malleable jobs.

	Avg. Turnaround	System Utilization	On-demand Jobs' Instant Start Ratio
Theta	15.6 hours	83.93%	22.69%
Cori	1.97 hours	80.27%	18.94%

N&PAA obtains the worst results on average job turnaround time and system utilization. Additionally, its malleable jobs' preemption ratio is noticeably higher than other mechanisms. The long average job turnaround time is caused by job starvation. Here, starvation means that jobs were preempted, but could not resume for a long period of time after preemption. Although on-demand jobs return their leased nodes to the lenders, the lenders might not resume immediately, because those on-demand jobs might need a portion of the preempted nodes and the rest are moved to the common resource pool. When the on-demand job is finished, the preempted job can only reclaim the nodes from the on-demand job and it has to wait until more nodes are available.

Observation 3. *To achieve higher system utilization and lower malleable jobs' preemption ratio, SPAA is preferred over PAA.*

All three SPAA methods largely reduce malleable jobs' preemption ratio, while slightly increasing the rigid jobs' preemption ratio. This is because SPAA attempts to find shrink options, which reduces malleable jobs' preemption ratio. Shrink, in general, has lower overhead and leads to fewer wasted computation cycles and therefore higher system utilization.

Observation 4. *To obtain lower average job turnaround time and lower rigid jobs' preemption ratio, PAA methods are recommended than SPAA methods, except N&PAA method.*

SPAA methods tend to prolong average job turnaround time, especially malleable jobs, because it reduces all running malleable jobs' sizes and prolongs their execution time. On the other hand, PAA affects fewer running jobs and the preempted jobs might resume when the on-demand job finishes. However, N&PAA is an exception. This is because CUA and CUP prepare some nodes for on-demand jobs before their arrival and PAA only needs to preempt small-sized running jobs upon on-demand job arrival. On the other hand, N&PAA is more likely to preempt large-sized running jobs, which are more difficult to reclaim their preempted nodes.

It is interesting to notice that PAA methods lead to slightly lower rigid jobs' preemption ratio than SPAA methods. Since

SPAA methods first try to shrink malleable jobs, the job sizes of running jobs are, on average, smaller than that of PAA methods. When the shrink option is not possible, SPAA methods need to preempt more running jobs, which causes slight increases in the rigid jobs' preemption ratio.

Observation 5. *CUA methods, in most cases, perform better than CUP methods.*

CUA methods achieve slightly lower average job turnaround time and slightly higher system utilization in most cases. CUA methods passively collect released nodes, while CUP methods proactively preempt some running jobs before on-demand job arrival. Therefore, CUA methods trigger fewer preemptions, leading to less resource waste and higher system utilization.

Observation 6. *CUA&PAA, CUA&SPAA, CUP&PAA, and CUP&SPAA encourage users to honestly declare their malleable jobs.*

Compared to rigid jobs, malleable jobs' turnaround time of these four methods is noticeably lower on Theta and slightly lower on Cori. For SPAA methods, although malleable jobs might need to shrink their sizes upon arrival of on-demand jobs, they are guaranteed to expand to their original sizes by reclaiming their released nodes when the on-demand job finishes. The malleability feature increases the chances of malleable jobs being chosen to execute, leading to lower average turnaround time compared to rigid jobs. The better job performance on malleable jobs discourages users from declaring malleable jobs as rigid jobs.

Observation 7. *N&SPAA method is a good option when rigid jobs need to achieve low average turnaround time.*

N&SPAA achieves the lowest rigid jobs' average turnaround time among the six methods. More importantly, rigid jobs yield similar or even lower average turnaround time compared to malleable jobs. When an on-demand job arrives, N&SPAA method first attempts to find shrink options. If there are viable shrink options, the selected malleable jobs will be shrunk and prolonged, while running rigid jobs are not impacted. Upon on-demand job arrival, N&SPAA requests more nodes than CUA&SPAA and CUP&SPAA. Therefore, N&SPAA has more noticeable adverse effects on malleable jobs than the other SPAA methods. Although N&SPAA does not provide strong incentives for malleable jobs, it is a good option for system administrators when rigid jobs have higher priority than malleable jobs.

Observation 8. *Malleable jobs' preemption ratio is noticeably higher than rigid jobs' preemption ratio.*

This is due to the fact that the preemption overheads of malleable jobs are lower than rigid jobs. In order to minimize wasted computation cycles caused by preemption, the running jobs are preempted in ascending order of their preemption overheads. Malleable jobs only waste their setup times. On the other hand, rigid jobs not only waste their setup times but also lose the computation after the latest checkpoints. It is interesting to notice that despite the higher preemption ratio, malleable jobs achieve lower average turnaround times because they are more likely to run by shrinking their sizes.

Observation 9. *All methods achieve extremely high on-demand jobs' instant start ratio.*

On-demand jobs represent 3%-15% of total capacity. On average, more than 98% of on-demand jobs start instantly. There is no significant difference in on-demand jobs' instant start ratio between the different methods. On-demand jobs fail to start immediately because the nodes used by running on-demand jobs plus this on-demand job exceed the system capacity. This metric is related to the on-demand jobs' submission pattern. Bursty on-demand job submission pattern could negatively affect their instant start ratio.

Observation 10. *Preemption has greater impact on workloads with large-sized jobs (e.g., Theta).*

Compared with FCFS/EASY, our proposed methods increase the average turnaround time by 30% on Theta workloads and 17% on Cori workloads. The main difference between Theta and Cori workloads is that the majority of Cori jobs are single-node jobs, while the most of Theta jobs are large-sized jobs. Small-sized jobs, especially single-node jobs, are more likely to resume after preemption and therefore have less impact on their turnaround times. On the other hand, large jobs tend to starve after preemption causing long turnaround times. On Theta, N&PAA tends to preempt large-sized jobs upon on-demand job arrival, which leads to extremely longer average turnaround time than other methods.

B. Impact of Accuracy of Advance Notice

Observation 11. *The performance of CUP methods highly relies on accuracy of advance notice. The more accurate advance notice, the better performance.*

CUP&PAA and CUP&SPAA methods achieve their best performance on W2, i.e., the workloads with the highest percentage of on-demand jobs with accurate advance notice. The accurate advance notice reduces the preemption ratio on both rigid and malleable jobs and therefore reduces wasted cycles and improves system utilization. The accurate advance notice also reduces average job turnaround time due to fewer interruptions during execution.

Observation 12. *The earlier the advance notice, the better the performance of CUA methods.*

CUA methods obtain the lowest average job turnaround time on W4, i.e., the workloads with the majority of on-demand jobs arrived late. W4 provides a longer period of time between advance notice and job actual arrival. As a result, CUA methods are more likely to collect nodes before the actual arrival of on-demand jobs, and thus decrease the chances of preempting or shrinking running jobs upon arrival of the on-demand jobs. In addition, by preparing more nodes for on-demand jobs before their arrival, it also slightly improves on-demand jobs' instant start ratio.

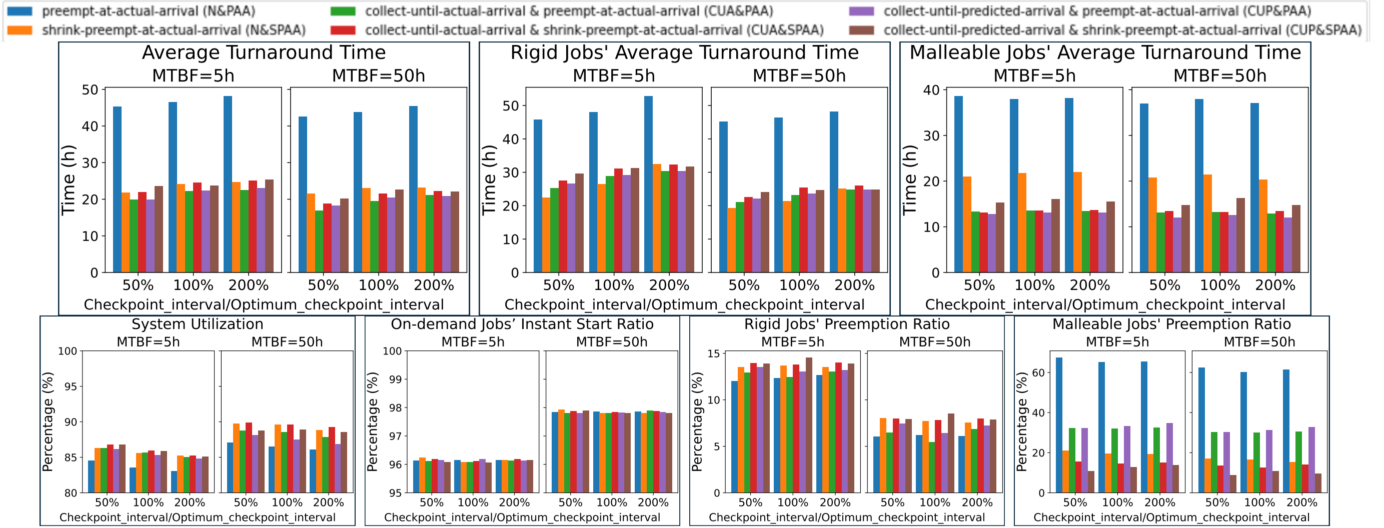

Fig. 8: Impact of rigid jobs' checkpointing frequency and failures on scheduling performance on Theta. For each metric, the left subfigure presents the results on the failure rate of MTBF $= 5h$ and the right subfigure presents the results on MTBF $= 50h$. For each subfigure, the x-axis presents the checkpoint interval relative to the optimum checkpoint interval. For example, 50% means rigid jobs make checkpoints twice as frequent as the optimum checkpoint frequency.

C. Impact of Checkpointing Frequency

Checkpointing is a technique providing fault tolerance for HPC systems. Different checkpointing frequencies not only impact the performance of hybrid workload scheduling, but also affect system's fault tolerance capability. In order to comprehensively evaluate the impact of checkpoints, we inject failures and conduct sensitive study on failures in this subsection. Mean time between failures (MTBF) and mean time to repair (MTTR) are two widely used metrics to describe HPC failures. MTBF measures the average time between failures, while MTTR is the average time to repair a failure on a HPC system. Based on the literature [43]–[45], production HPC systems' MTBF is typically between 5 and 50 hours and MTTR is approximately 6 hours. We conduct two sets of experiments with two levels of MTBF, i.e., 5 hours and 50 hours. Take MTBF $= 5h$ for example: node failures are randomly injected to simulated systems at the average rate of 5 hours per failure. After a failure, a node will be down for a period of time T_d to simulate the repair time. T_d is a randomly generated number with the mean time of 6 hours. The job running on failed node needs to either resume from the latest checkpoint or restart from the beginning if no checkpoint had been made. Figure 8 presents the scheduling results on Theta under different checkpointing frequencies and MTBF. Cori results lead to similar observations. Due to space limitation, we only present Theta results in the rest of this section.

Observation 13. *To achieve better rigid job performance and system performance, we suggest that rigid jobs take more frequent checkpoints than the optimum checkpointing frequency.*

All methods benefit from the more frequent checkpointing frequency. More frequent checkpoints can reduce rigid jobs' turnaround time and also improve system utilization. Daly's optimum checkpointing frequency is designed for fault tolerance [38]. However, the interruptions caused by failures are obviously much less frequent than the preemption caused by draining nodes for on-demand jobs. Therefore, increasing checkpointing frequency reduces rigid jobs' lost computation and thus reduces their turnaround time. This also helps improve system utilization by reducing preemption overheads.

Observation 14. *Failures have negative impact on system utilization, on-demand jobs instant start ratio, jobs preemption ratio and rigid jobs average turnaround time. More frequent checkpoints can mitigate negative effects on rigid jobs.*

When reducing MTBF from 50 to 5 hours, we observe increases in rigid jobs' average turnaround time and jobs preemption ratio. We also observe decreases in on-demand jobs' instant start ratio and system utilization. Failures on running rigid or malleable jobs cause higher preemption ratio, while failures on running on-demand jobs lead to lower instant start ratio. More job preemptions also lead to higher preemption overheads and thus lower system utilization. When nodes running rigid or on-demand jobs fail, these jobs cannot reduce their sizes by using the remaining functional nodes leading to prolonged turnaround times. More frequent checkpoints can mitigate the negative effects on rigid jobs' average turnaround time, because the failed rigid jobs need to do less re-computation.

D. Impact of Malleable Job Sizes

Figure 9 presents the scheduling results on Theta under different malleable job size ranges.

Observation 15. *Reducing the job size range of malleable jobs has adverse effects on both system-level and user-level scheduling performance, especially on SPAA methods.*

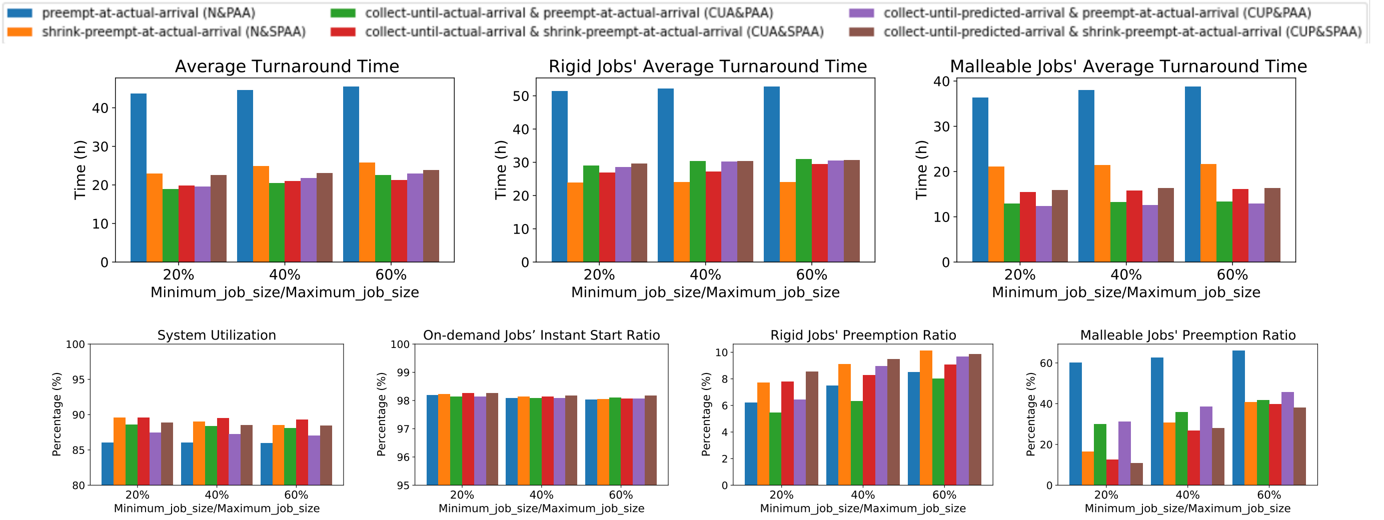

Fig. 9: Impact of malleable job sizes on scheduling performance on Theta. 20% means malleable job's minimum size is twenty percent of its maximum size. The larger the minimum size is, the less flexible the malleable job is.

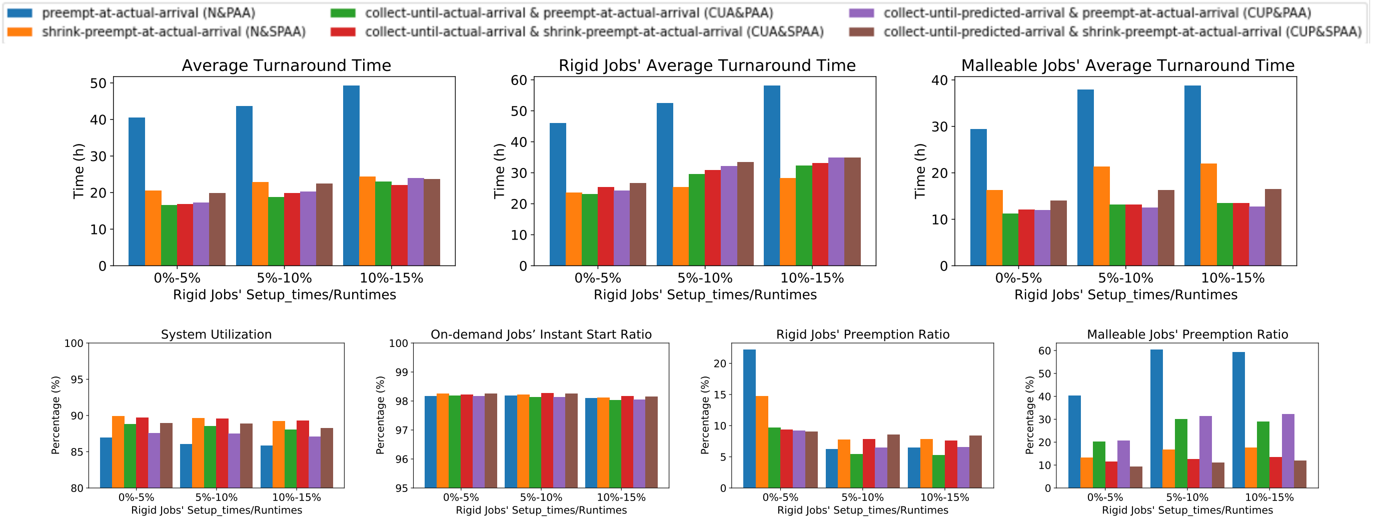

Fig. 10: Impact of rigid jobs' setup overheads on scheduling performance on Theta. 0%-5% means rigid job's overhead is randomly chosen between 0% to 5% of its runtime.

The increases in the minimum malleable job sizes reduce the malleable jobs' size range, causing decreased chances of shrinking malleable jobs. Therefore, it increases the preemption ratio of both malleable and rigid jobs, which leads to slight decreases in system utilization. The changes in minimum malleable job sizes have a greater impact on SPAA methods, because SPAA cannot shrink malleable jobs to smaller sizes for on-demand jobs.

E. Impact of Setup Overheads

Figure 10 presents the scheduling results on Theta under different rigid jobs' setup overheads.

Observation 16. *The lower the setup overhead is, the better the scheduling performance is.*

When a preempted job resumes, it takes some time to set up. Therefore, the higher setup overhead means the higher preemption overhead and thus the higher average job turnaround time. Additionally, higher setup overhead decreases system utilization by wasting more computation cycles for setup.

Observation 17. *If rigid jobs' setup overheads reduce to the amount which is similar to malleable jobs' setup overhead, the rigid jobs' preemption ratio will largely increase.*

In our experiments, malleable jobs' setup overhead is set to 0%-5%. If rigid jobs' setup overhead is reduced to 0%-5%, we notice the obvious increases in rigid jobs' preemption ratio and decreases in malleable jobs' preemption ratio, especially on PAA methods. This is because we order running jobs based on their preemption overheads. If we preempt rigid jobs immediately after their checkpoints, their preemption overhead is similar to the overhead of preempting malleable jobs. As a result, rigid jobs' preemption ratio is increased.

VI. Conclusion

In this paper, we have defined and modeled HPC hybrid workload scheduling problem on a single HPC system. We have proposed six mechanisms to reconcile the demands from on-demand, rigid, and malleable applications. We have thoroughly evaluated our mechanisms based on two production HPC system traces. By exploring how different mechanisms behave under various configurations and workloads, we have found that it is feasible to accommodate rigid, malleable, and on-demand jobs on a single HPC system via co-scheduling mechanisms. Additionally, we have provided several key insights on the mechanisms. In particular, these mechanisms significantly improve system utilization. In terms of on-demand jobs, the mechanisms boost their instant start ratio. Users wait less time by declaring their jobs as malleable jobs. Although preemptions slightly increase rigid jobs' turnaround time, more frequent checkpoints can mitigate the negative impact due to lower preemption overhead. To the best of our knowledge, this co-scheduling study is the first of its kind. We believe that our findings will be useful in understanding the trade-offs of co-scheduling these three types of jobs under various situations.

Acknowledgment

This work is supported in part by US National Science Foundation grants CCF-2109316, CNS-1717763, and CCF-2119294 and U.S. Department of Energy, Office of Science, under contract DE-AC02-06CH11357. Job logs from the Cori system were provided by the National Energy Research Scientific Computing Center operated under Contract No. DE-AC02-05CH11231.

References

[1] Aurora. https://www.alcf.anl.gov/aurora/.
[2] Summit. https://www.olcf.ornl.gov/summit/.
[3] M. Jette, A. Yoo, and M. Grondona. SLURM: Simple Linux Utility for Resource Management. In *JSSPP*, 2003.
[4] Moab. http://www.adaptivecomputing.com/ products/hpc-products/moab-hpc-basic-edition/.
[5] PBS Professional. http://www.pbsworks.com/.
[6] Cobalt. https://www.alcf.anl.gov/cobalt-scheduler.
[7] F. Liu, K. Keahey, P. Riteau, and J. Weissman. Dynamically Negotiating Capacity between On-Demand and Batch Clusters. SC, 2018.
[8] A. Maurya, B. Nicolae, I. Guliani, and M. Rafique. CoSim: A Simulator for Co-Scheduling of Batch and On-Demand Jobs in HPC Datacenters. In *DS-RT*, 2020.
[9] P. Beckman, S. Nadella, N. Trebon, and I. Beschastnikh. Spruce: A system for supporting urgent high-performance computing. *International Federation for Information Processing*, 2007.
[10] M. Agung, Y. Watanabe, H. Weber, R. Egawa, and H. Takizawa. Preemptive parallel job scheduling for heterogeneous systems supporting urgent computing. *IEEE Access*, 2021.
[11] S. H. Leong and D. Kranzlmuller. A case study - cost of preemption for urgent computing on supermuc. In *HiPC*, 2015.
[12] P. Lemarinier, K. Hasanov, S. Venugopal, and K. Katrinis. Architecting malleable mpi applications for priority-driven adaptive scheduling. *Proc. of the European MPI Users' Group Meeting*, 2016.
[13] M. Anderson, S. Smith, N. Sundaram, M. Capotă, Z. Zhao, S. Dulloor, N. Satish, and T. Willke. Bridging the Gap between HPC and Big Data Frameworks. *Proc. VLDB Endow*, 2017.
[14] M. Salim, T.Uram, T. Childers, Balaprakash P, V. Vishwanath, and M. Papka. Balsam: Automated Scheduling and Execution of Dynamic, Data-Intensive HPC Workflows, 2019.
[15] T. Carroll and D. Grosu. Incentive Compatible Online Scheduling of Malleable Parallel Jobs with Individual Deadlines. In *International Conference on Parallel Processing*, 2010.
[16] H. Sun, Y. Cao, and W. Hsu. Fair and Efficient Online Adaptive Scheduling for Multiple Sets of Parallel Applications. In *IEEE 17th International Conference on Parallel and Distributed Systems*, 2011.
[17] A. Souza, M. Rezaei, E. Laure, and J. Tordsson. Hybrid Resource Management for HPC and Data Intensive Workloads. In *CCGRID*, 2019.
[18] M. Chadha, J. John, and M. Gerndt. Extending SLURM for Dynamic Resource-Aware Adaptive Batch Scheduling. In *HiPC*, 2020.
[19] J. Uisson, O. Sonmez, H. Mohamed, W. Lammers, and D. Epema. Scheduling Malleable Applications in Multicluster Systems. In *IEEE International Conference on Cluster Computing*, 2007.
[20] D. Kumar, Z. Shae, and H. Jamjoom. Scheduling Batch and Heterogeneous Jobs with Runtime Elasticity in a Parallel Processing Environment. In *IPDPS PhD Forum*, 2012.
[21] J. Hungershofer. On the combined scheduling of malleable and rigid jobs. In *16th Symposium on Computer Architecture and High Performance Computing*, 2004.
[22] S. Prabhakaran, M. Neumann, S. Rinke, F. Wolf, A. Gupta, and L. Kale. A Batch System with Efficient Adaptive Scheduling for Malleable and Evolving Applications. In *IPDPS*, 2015.
[23] G. Ananthanarayanan, C. Douglas, R. Ramakrishnan, S. Rao, and I. Stoica. True Elasticity in Multi-Tenant Data-Intensive Compute Clusters. In *SoCC*, 2012.
[24] Theta. https://www.alcf.anl.gov/theta.
[25] Cori. https://docs.nersc.gov/systems/cori/.
[26] D. Feitelson and L. Rudolph. Towards Convergence in Job Schedulers for Parallel Supercomputers. IPPS, 1996.
[27] I. Raicu, Zhao Zhang, M. Wilde, I. Foster, P. Beckman, K. Iskra, and B. Clifford. Toward Loosely Coupled Programming on Petascale Systems. In *SC*, 2008.
[28] I. Sadooghi, J. H. Martin, T. Li, K. Brandstatter, K. Maheshwari, T. Ruivo, G. Garzoglio, S. Timm, Y. Zhao, and I. Raicu. Understanding the Performance and Potential of Cloud Computing for Scientific Applications. *IEEE Transactions on Cloud Computing*, 2017.
[29] Q. He, S. Zhou, B. Kobler, D. Duffy, and T. McGlynn. Case Study for Running HPC Applications in Public Clouds. HPDC, 2010.
[30] W. Allcock, P. Rich, Y. Fan, and Z. Lan. Experience and Practice of Batch Scheduling on Leadership Supercomputers at Argonne. In *JSSPP*, 2017.
[31] A. Mu'alem and D. Feitelson. Utilization, Predictability, Workloads, and User Runtime Estimates in Scheduling the IBM SP2 with Backfilling. *TPDS*, 2001.
[32] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. NSDI, 2011.
[33] Kubernetes. https://kubernetes.io/.
[34] P. Ambati, N. Bashir, D. Irwin, and P. Shenoy. Waiting Game: Optimally Provisioning Fixed Resources for Cloud-Enabled Schedulers. SC, 2020.
[35] Y. Fan, P. Rich, W. Allcock, M. Papka, and Z. Lan. Trade-Off Between Prediction Accuracy and Underestimation Rate in Job Runtime Estimates. In *CLUSTER*, 2017.
[36] AWS Spot Instance interruptions. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/spot-interruptions.html.
[37] Job Scheduling Policy for Theta. https://www.alcf.anl.gov/support-center/theta/job-scheduling-policy-theta.
[38] J. Daly. A Higher Order Estimate of the Optimum Checkpoint Interval for Restart. Future Generation Computer Systems, 2006.
[39] A. Moody, G. Bronevetsky, K. Mohror, and B. Supinski. Design, Modeling, and Evaluation of a Scalable Multi-level Checkpointing System. SC, 2010.
[40] S. Di, M. Bouguerra, L.A. Bautista-Gomez, and F. Cappello. Optimization of Multilevel Checkpoint Model for Large Scale HPC Applications. IPDPS, 2014.
[41] L. Yu, Z. Zhou, Y. Fan, M. Papka, and Z. Lan. System-wide Trade-off Modeling of Performance, Power, and Resilience on Petascale Systems. In *The Journal of Supercomputing*, 2018.
[42] CQSim Github Repository. https://github.com/SPEAR-IIT/CQSim.
[43] D. Tiwari, S. Gupta, and S. S. Vazhkudai. Lazy Checkpointing: Exploiting Temporal Locality in Failures to Mitigate Checkpointing Overheads on Extreme-Scale Systems. In *DSN*, 2014.
[44] B. Schroeder and G. A. Gibson. A Large-Scale Study of Failures in High-Performance Computing Systems. *TDSC*, 2010.
[45] T. J. Hacker, F. Romero, and C. D. Carothers. An Analysis of Clustered Failures on Large Supercomputing Systems. *JPDC*, 2009.