CoSim: A Simulator for Co-Scheduling of Batch and On-Demand Jobs in HPC Datacenters

Avinash Maurya, Bogdan Nicolae, Ishan Guliani, M Mustafa Rafique

CoSim: A Simulator for Co-Scheduling of Batch and On-Demand Jobs in HPC Datacenters

Avinash Maurya*, Bogdan Nicolae†, Ishan Guliani*, M. Mustafa Rafique*
*Rochester Institute of Technology, USA
†Argonne National Laboratory, USA
Email: *{am6429, ig5859, mrafique}@cs.rit.edu; †bnicolae@anl.gov

***Abstract*—The increasing scale and complexity of scientific applications are rapidly transforming the ecosystem of tools, methods, and workflows adopted by the high-performance computing (HPC) community. Big data analytics and deep learning are gaining traction as essential components in this ecosystem in a variety of scenarios, such as, steering of experimental instruments, acceleration of high-fidelity simulations through surrogate computations, and guided ensemble searches. In this context, the batch job model traditionally adopted by the supercomputing infrastructures needs to be complemented with support to schedule opportunistic on-demand analytics jobs, leading to the problem of efficient preemption of batch jobs with minimum loss of progress. In this paper, we design and implement a simulator, *CoSim*, that enables on-the-fly analysis of the trade-offs arising between delaying the start of opportunistic on-demand jobs, which leads to longer analytics latency, and loss of progress due to preemption of batch jobs, which is necessary to make room for on-demand jobs. To this end, we propose an algorithm based on dynamic programming with predictable performance and scalability that enables supercomputing infrastructure schedulers to analyze the aforementioned trade-off and take decisions in near real-time. Compared with other state-of-art approaches using traces of the Theta pre-Exascale machine, our approach is capable of finding the optimal solution, while achieving high performance and scalability.**

***Index Terms*—High-performance computing, batch job preemption, job checkpointing**

I. Introduction

Big data analytics and deep learning are rapidly gaining traction both in the industry and scientific computing. A key driver for this trend has been the unprecedented accumulation of big data, which exposes plentiful learning opportunities thanks to its massive size and variety. Unsurprisingly, there has been a significant interest to adopt deep learning at a very large scale on supercomputing infrastructures in a wide range of scientific areas, e.g., fusion energy science [1], computational fluid dynamics [2], lattice quantum chromodynamics [3], virtual drug response prediction [4], and cancer research [5].

One of the main use cases of big data analytics and deep learning in scientific computing is to use them as a tool to complement high-performance computing (HPC) simulations running on supercomputing infrastructures in a variety of scenarios: steering of experimental instruments (e.g., calibrate scientific instruments in real-time to correct anomalies in experimental data and/or refocus dynamically on areas of interest), acceleration of high-fidelity simulations through surrogate computations (e.g., under the right circumstances, expensive steps of an HPC simulation can be replaced with faster deep learning predictions), guided ensemble searches (e.g., when running a set of simulations to find a molecule that docks to a protein, deep learning can be used to predict the next most promising simulations to try next).

These scenarios require running opportunistic on-demand jobs when certain conditions are triggered, e.g., an analytics job that looks for anomalies in the experimental data collected by the instrument, a deep learning training and/or inference. These jobs need to start within a given deadline, often in the order of minutes. Failure to start them by the given deadline leads to a missed opportunity (e.g., it's too late to calibrate the instrument) and/or incur a performance penalty (e.g., idle simulations that wait for the next deep learning prediction or otherwise take alternative suboptimal decisions). On the other hand, HPC datacenters traditionally adopt a batch job scheduling model where users request compute and accelerator (e.g., GPU) resources of the datacenters for the required amount of time (wall time), while the scheduler decides when to run each job based on various trade-offs such as the need to maximize the utilization of machines, and job priority. Popular HPC datacenter schedulers, e.g., SLURM (Simple Linux Utility Resource Manager) [6], COBALT [7], and TORQUE (Terascale Open-source Resource and QUEue manager) [8], cannot co-schedule batch jobs with opportunistic on-demand jobs.

A naive solution to address this problem could simply reserve a set of nodes for on-demand jobs and use the rest of the nodes for batch jobs. Although applied in practice, such a solution is not desired as it is hard to predict how many nodes are needed by the on-demand jobs. Using too few nodes for on-demand jobs leads to missed opportunities, whereas, using too many nodes leads to idle nodes and slow progress of the batch jobs. Furthermore, even if such predictions were perfect, there may be significant fluctuations in datacenter utilization patterns that make it hard to dynamically move the nodes back and forth between on-demand and batch queues. At the other extreme, an alternative naive solution could simply use all nodes for batch jobs and start killing batch jobs to make room for on-demand jobs when needed. This solution does not lead to missed opportunities but may incur significant overhead on the batch jobs due to loss of progress.

In this paper, we propose an alternative solution to address these challenges that relies on checkpointing for suspending and resuming batch jobs, if required, to make room for time-

978-1-7281-7343-6/20/$31.00 ©2020 IEEE

sensitive on-demand jobs, thereby minimizing the amount of lost progress by the batch jobs. To this end, we introduce *CoSim*, a simulation framework that aims to identify the optimal combination of jobs that should be either checkpointed or killed to free a fixed number of nodes that are required to run the on-demand job. Unlike other approaches, *CoSim* simulates all outcomes resulting from a variable deadline up to the given maximum in a single pass, thereby eliminating the need to run separate simulations for each fixed deadline. Using this approach, the scheduler can make more informed decisions by considering the various trade-offs arising from delaying the start of the on-demand jobs and losing progress on the batch jobs. Specifically, we make the following contributions in this paper:

- We formulate the problem statement, introducing a series of assumptions and general considerations for simulating the outcomes of checkpointing batch jobs to vacate nodes for running on-demand jobs (Section II).
- We introduce a series of design principles and an algorithm based on the dynamic programming to find the optimal combination of batch jobs that incurs the minimum loss of progress while satisfying the deadline of the on-demand jobs. Our algorithm produces an optimal solution for every possible deadline up to a given maximum in a single pass (Section III).
- We evaluate our approach in a series of experiments using three scenarios extracted from the batch job traces of Argonne's Theta pre-Exascale machine. We compare our approach with an exhaustive search based on backtracking and a greedy approach. The results show significant performance and scalability improvement as compared to backtracking, as well as a significant improvement in the quality of the solution compared as compared to the greedy approach (Section IV).

II. Problem Formulation

The problem of co-scheduling batch jobs with opportunistic on-demand jobs in an HPC datacenter can be formulated as follows. Let's assume that N batch jobs are running at time t_0, and each of these jobs is characterized by the tuple $\langle id, jn, loss, tckpt \rangle$, where id is a unique identifier of the job, jn is the number of compute nodes the batch job is running on, $loss$ quantifies the amount of lost progress if the job is killed (e.g., node-hours since the last checkpoint or since the beginning if no checkpoint was taken), $tckpt$ is the time required to checkpoint the job to successfully suspend its execution without loss of progress.

Given K nodes that need to be released not later than t_0+T (for the purpose of starting opportunistic on-demand jobs), the goal is to find all optimal subsets of batch jobs $S_i \subset N$ and corresponding killing or checkpointing strategy for all $t_0 < i < t_0 + T$. A subset S_i is optimal if it satisfies the following properties simultaneously: (1) at least K nodes are released by the deadline t_0+i; (2) the accumulated loss of work due to killing jobs is minimized; (3) if there are multiple subsets S_i for which the accumulated loss due to job kills is minimized, then prefer the subset for which the checkpointing overhead is minimized. We refer to this as *the eviction problem*.

Using the subsets S_i and corresponding strategy, an HPC datacenter scheduler can simulate the outcome of multiple hypothetical scenarios with a variable deadline i corresponding to the trade-off between maximizing the value of the on-demand jobs and minimizing the loss of the batch jobs.

We note that while we formulate the problem of HPC datacenters, a similar formulation can be done for opportunistic jobs in cloud computing architectures where there is an upper bound on elasticity, e.g., the user cannot afford to run on more than a fixed amount of virtual machines (VMs) at a time and must evict existing jobs if necessary. Without loss of generality, *CoSim* can be applied in such scenarios as well.

III. Design Principles and Approach

This section introduces the high-level design principles of our proposed approach and explains aspects related to the checkpointing model and exploration algorithm that implement these design principles.

A. Design principles

CoSim is based on the following design principles:

1) Mix of system-level and application-level checkpointing: We differentiate between system-level and application-level checkpointing, because they present an interesting trade-off: system-level checkpointing techniques, such as DMTCP [9], are application-agnostic and can be performed at any moment t_0. However, they involve large checkpoint sizes because the entire memory space of all application processes needs to be persisted to a stable storage, e.g., a parallel file system (PFS). Therefore, system-level checkpointing may take a long time to complete. On the other hand, application-level checkpointing is typically performed by HPC applications regularly using either a custom solution or a checkpointing library, such as, VELOC [10]. In this case, the checkpoint size is smaller as each application process needs to save only the critical data structures needed for a restart and therefore faster to write to the stable storage. However, it is necessary to wait for the application to reach a moment $t_1 > t_0$ when it is safe to checkpoint. Depending on how far away t_1 is from t_0 and how much larger a system-level checkpoint is compared with an application-level checkpoint, one or the other may be faster. Furthermore, it is important to note that even if the system and application-level checkpointing overheads are equal, it is still important to choose the application-level checkpoint over the system-level checkpoint, because using application-level checkpoint enables the batch job to make additional progress during the interval (t_0, t_1). We incorporate such considerations in our simulator.

2) Simultaneous exploration of the full on-demand deadline range: As discussed in Section II, our goal is to solve the eviction problem for all deadlines in the range $(t_0, t_0 + T)$, because the scheduler needs to consider the trade-off between delaying the on-demand jobs, which may lead to lower quality of the results due to slow reaction time, and losing progress

of the batch jobs. Thus, a naive strategy would be to iterate over all deadlines i in range $(t_0, t_0 + T)$ and solve the problem independently for each i. However, such a strategy is sub-optimal, because the problems resulting from fixing all deadlines i in range $(t_0, t_0 + T)$ have identical inputs except for the deadline i, therefore they may be decomposed into sub-problems that are shared across several instances and thus need to be solved only once. Our approach leverages this observation to construct an algorithm based on dynamic programming that is capable of taking advantage of such decompositions to solve all deadlines in a single pass. This algorithm is explained in Section III-C.

3) Polynomial response time: A key requirement for exploring the full on-demand deadline range is to ensure fast response time so that the scheduler can decide quickly, preferably at moment t_0, about the jobs that must be checkpointed to the PFS to run the incoming on-demand jobs. Thus, an algorithm that is not polynomial in any variable, such as, the number of jobs N, maximum deadline T, or the number of nodes to be released K, will lead to unacceptable response time, considering that modern HPC datacenters routinely run several batch jobs simultaneously and may need to release a large number of nodes for on-demand jobs to accommodate bursts of opportunistic events. Therefore, our proposed solution is designed to satisfy such constraints, and delivers response times in the order of milliseconds or less.

B. Loss and checkpointing model

We estimate the loss incurred by killing a batch job as the number of node-hours that have elapsed since its last application-level checkpoint until t_0, the moment when the nodes need to be evicted to make room for the on-demand jobs. This is based on a configurable interval that can be independently adjusted for each job in our simulator. In practice, the interval is fixed based on empirical observations, e.g., every hour, because the checkpoints are used both to survive failures and to record intermediate results. However, if checkpoints are only used for fault tolerance, then an optimal checkpointing interval can be computed [11].

In order to simulate alternatives that mix application-level checkpointing with system-level checkpointing, we consider the time to checkpoint each batch job:

$$tckpt = max(\sum_{i=1}^{jn} sckpt(i)/B_a, max_{i=1}^{jn}(sckpt(i)/B_c)) \quad (1)$$

where jn is the number of nodes occupied by the batch job, B_a is the aggregated I/O bandwidth of the PFS, B_c is the maximum I/O bandwidth of a compute node, and $sckpt(i)$ is the size of the checkpoint on node $i \in [1 \dots jn]$. The intuition behind this is that the checkpointing time is bounded either by the maximum aggregated bandwidth of the PFS or the slowest node (if the nodes do not consume the maximum aggregated bandwidth).

In the case of application-level checkpointing, we must wait for the next checkpoint to happen, which introduces a delay in addition to $tckpt$. Therefore, the application-level checkpointing duration is $ta = tckpt + delay$, where $delay$ is the difference between the next scheduled checkpoint and t_0. Since system-level checkpointing can be performed instantly at t_0, its duration, ts, will be equal to $tckpt$, i.e., $ts = tckpt$. However, the two approaches will have a different checkpoint size on each node, resulting in the trade-off that is explained in Section III-A.

In a typical HPC datacenter, the size of each batch job is usually large enough to saturate the aggregated I/O bandwidth of the PFS. Therefore, we consider a simple checkpointing model where the jobs are checkpointed serially. In this case, the total time required for checkpointing a set of batch jobs is the sum of their corresponding ta or ts. In fact, under such circumstances, checkpointing multiple batch jobs in parallel would perform worse than checkpointing the batch jobs serially, because of over-subscribing the aggregated I/O bandwidth of the PFS. Nevertheless, we note that our model can be further refined to simulate concurrent checkpointing of the batch jobs in the case of small jobs that do not saturate the aggregated I/O bandwidth of the PFS.

C. Exploration algorithm

In this section we propose a dynamic programming algorithm based on the aforementioned design principles.

The key observation that inspires our algorithm is the fact that the eviction problem is related to the discrete backpack problem: given N items, where W_i and V_i represent the weight and value of the i^{th} item, fill a backpack that can carry a maximum weight K such that the combined value of all items is maximized without overflowing K. By analogy, we can consider the batch jobs as items and the backpack as the set of nodes where the jobs are running. This problem has a simple dynamic programming decomposition: maximum value for N items is the greater of: (1) the maximum value for $N-1$ items and capacity K (excludes item N); (2) V_N plus the maximum value obtained for $N-1$ items and capacity $K-W_N$ (includes item N). By solving this decomposition recursively and applying memoization techniques, a runtime complexity of $O(N \cdot K)$ can be achieved. Note that this decomposition solves the problem not only for a backpack of capacity K, but at the same time for all backpacks of capacity i such that $0 < i \leq K$.

Starting from this observation, we adopt a similar strategy but with two important differences. First, we need to free at least K nodes, which means that the optimal solution may involve more than K nodes. Therefore, we need to consider up to M nodes, where M is the number of nodes occupied by all batch jobs at the moment t_0. Second, the eviction problem introduces a new dimension in the decomposition, i.e., the deadline T to start the on-demand jobs. Specifically, it is not enough to release at least K nodes within a deadline T when considering $N-1$ batch jobs and then try for the N^{th} batch job all four alternatives, i.e., ignore, kill, application-level checkpoint, and system-level checkpoint, because the optimal solution for $N-1$ batch jobs may get close to the deadline

Algorithm 1: Dynamic programming algorithm to free K nodes within a range of deadlines $[0 \dots T]$ with minimal loss of compute progress.

Input: List J of N batch jobs running at t_0, K, T
Output: List of job eviction strategies S_i, $0 < i < T$

$a[0,0] \leftarrow 0$
$u[0,0] \leftarrow \emptyset$
$M \leftarrow 0$
for $(id, jn, loss, ts, ta) \in J$ **do**
  $M \leftarrow M + jn$
for $(id, jn, loss, ts, ta) \in J$ **do**
  $b \leftarrow a$
  $v \leftarrow u$
  for $(n,t) \in a$ **do**
    if $b[n+jn,t] > a[n,t] + loss$ **then**
      $b[n+jn,t] \leftarrow a[n,t] + loss$
      $v[n+jn,t] \leftarrow u[n,t] \cup \{(id, \text{“}kill\text{”})\}$
    if $t + ta <= T \wedge b[n+jn, t+ta] > a[n,t]$ **then**
      $b[n+jn,t+ta] \leftarrow a[n,t]$
      $v[n+jn,t+ta] \leftarrow u[n,t] \cup \{(id, \text{“}app\text{”})\}$
    if $t + ts <= T \wedge b[n+jn, t+ts] > a[n,t]$ **then**
      $b[n+jn,t+ts] \leftarrow a[n,t]$
      $v[n+jn,t+ts] \leftarrow u[n,t] \cup \{(id, \text{“}sys\text{”})\}$
  $a \leftarrow b$
  $u \leftarrow v$
for $i \in [0 \dots T]$ **do**
  $(x,y) \leftarrow argmin(a[x = K \dots M, y = 0 \dots i])$
  $Result[i] \leftarrow (a[x,y], u[x,y])$
return $Result$

T, thereby limiting the set of valid choices for job N (e.g., no further checkpointing is possible within T).

As a consequence, we propose a two-dimensional decomposition based on both the number of nodes and the deadline. We denote with the tuple $\langle jn_N, loss_N, ts_N, ta_N \rangle$ the number of nodes, loss of progress due to job killing, system-level checkpointing duration, and application-level checkpointing duration for job N. Then, the minimum loss for N jobs, M nodes, and deadline T denoted as $a[N, M, T]$ is the lesser of: (1) ignore job N, i.e., $a[N-1, M, T]$; (2) kill job N, i.e., $loss_N + a[N-1, M-jn_N, T]$; (3) take an application-level checkpoint of job N, i.e., $a[N-1, M-jn_N, T-ta_N]$; and (4) take a system-level checkpoint of job N, i.e., $a[N-1, M-jn_N, T-ts_N]$. Algorithm 1 presents our approach to solve this decomposition with a runtime of $O(N \cdot M \cdot T)$. The output of this algorithm is a list S_i for $0 < i < T$, where S_i is the set of jobs to be evicted using an optimal strategy such that the compute loss is minimized, and, in case of multiple solutions with minimal compute loss, the checkpointing overhead is minimized too.

We note that Algorithm 1 uses a temporary minimum loss matrix b and a corresponding solution matrix v to hold the updates resulting from considering all alternatives for job id. This is needed in order to avoid repeatedly selecting the same id in subsequent decompositions. Furthermore, the application-level checkpointing strategy takes precedence over the system-level checkpointing during the updates, and thus becomes a preferred choice in the case of equal loss and checkpointing time.

Another important observation is that Algorithm 1 solves the eviction problem not only for at least K nodes, but the entire node spectrum $[0 \dots M]$. This enables the scheduler to consider more advanced trade-offs for on-demand jobs, such as running the on-demand jobs with more or less than K requested nodes, which can be used to dynamically adjust the latency and/or the quality of the on-demand results. Such trade-offs can be incorporated at no additional simulation cost using our proposed approach.

IV. Performance Evaluation

To evaluate our proposal, we study the traces of Argonne's *Theta* pre-Exascale machine and extract three representative scenarios that create a challenging situation with respect to the eviction problem: most of the nodes are occupied by a relatively large number of batch jobs, leading to many possible combinations that need to be explored. For each scenario, we augment the traces with additional data that enables us to apply our model in order to extract the parameters of each batch job: compute loss and application-level/system-level checkpointing duration. We then compare our dynamic programming algorithm with two other approaches: a greedy algorithm (linear complexity) and a backtracking algorithm that performs an exhaustive search (exponential complexity). For the rest of this section, we introduce the methodology of our proposal and discuss the results of the comparison.

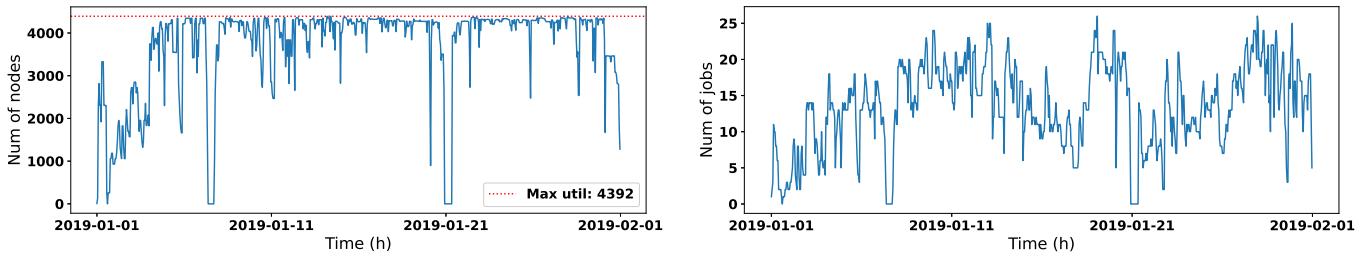
A. Batch job traces

In this paper, we consider the case of Argonne's *Theta* supercomputer, a 11.69 petaflops pre-Exascale Cray XC40 system based on the second-generation KNL Intel Xeon Phi 7230 SKU. The system has 4392 nodes, each equipped with 64 core processors (256 hardware threads), 16 GB of high-bandwidth MCDRAM (300-450 GB/s), 192 GB of main memory (DDR4 RAM, 20 GB/s), and a 128 GB SSD (700 MB/s). The interconnect topology is based on Dragonfly with a total bisection bandwidth of 7.2 TB/sec. Durable storage is provided by a Lustre parallel file system that is accessible to the compute nodes through a POSIX mount point. The total aggregated bandwidth is 250 GB/s.

First, we study the *DIM_JOB_COMPOSITE* trace[1] of batch jobs executed on Theta between 2017 and 2019. Specifically, we extract for each job the required fields pertaining to the runtime (start time, execution time) and the number of nodes. Then, we aggregate this information to obtain the number of batch jobs and the number of nodes utilized by the batch jobs per time unit. We focus in particular on the year 2019, which reflects the most recent utilization pattern: a total of 91,217 batch jobs were executed during the entire year.

We zoom on the node utilization (Figure 1a) and the number of jobs (Figure 1b) per hour during January 2019. A similar

[1] https://reports.alcf.anl.gov/data/theta.html



(a) Node utilization

(b) Number of jobs

Fig. 1: Trace analysis for January 2019.

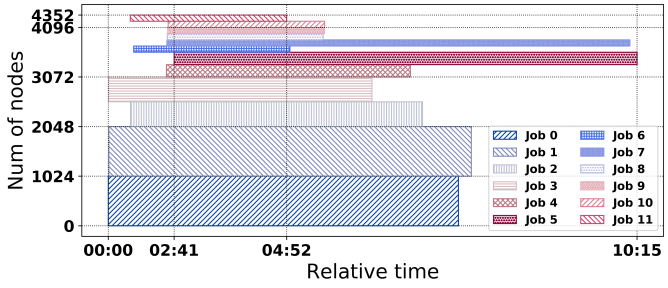


Fig. 2: *Scenario-1*: 12 batch jobs running on 4352 nodes.

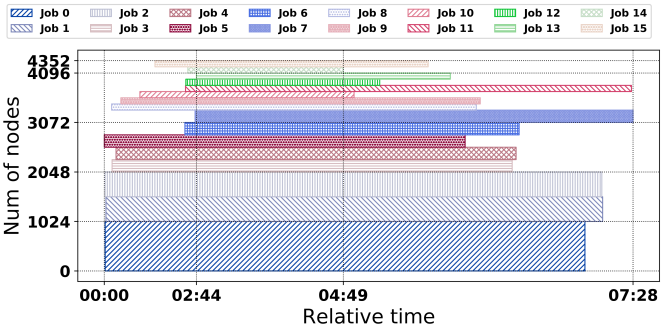


Fig. 3: *Scenario-2*: 16 batch jobs running on 4352 nodes.

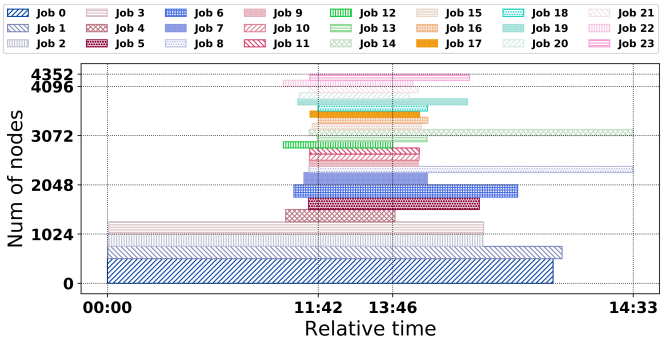


Fig. 4: *Scenario-3*: 24 batch jobs running on 4352 nodes.

pattern can be observed for the rest of the year. These figures reveal several interesting observations:

- During the entire year, all 4392 nodes were occupied for only around 1.5 days. However, the most frequent number of occupied nodes is 4352, which is close to the maximum capacity and appears for a total of 34 days. Therefore, the likelihood of having to run on-demand jobs when the machine runs batch jobs close to full capacity is very high.
- When the machine is operating close to capacity (4352 occupied nodes), the number of jobs is relatively high, peaking at around 25 jobs.
- About 61% of the batch jobs reported an execution time of less than 30 minutes. We consider these batch jobs *expendable*, such that killing them incurs negligible loss.

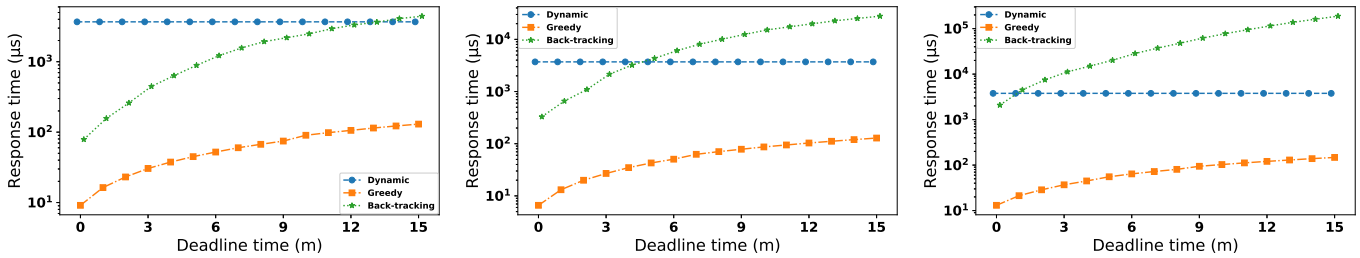
Based on these observations, we construct three representative scenarios, each of which occupies 4352 nodes at moment t_0 and a variable number of jobs: 12, 16, 24. We deliberately avoid expendable jobs in these scenarios (i.e., no expendable job is running at moment t_0) in order to create a challenging situation where all jobs may incur a significant loss of node-hours. The scenarios are illustrated in Figure 2, Figure 3 and Figure 4. The regions of interest during which all jobs are running are marked with between two vertical timestamps relative to the beginning of the earliest job. For example, Figure 2 captures a scenario of 12 jobs running for a total of 10 hours and 15 minutes, where, all batch jobs overlap for about 2 hours, i.e., from 02:41 to 04:52. The moment t_0 is chosen within these regions of interest.

B. Augmentation of the traces with checkpointing parameters

The *DIM_JOB_COMPOSITE* trace does not capture any information about the checkpointing behavior of the batch jobs. Lacking such information, we augment the scenarios with a series of synthetically generated checkpointing information based on empirical observations. Specifically, we assume that all batch jobs conduct application-level checkpoints at an hourly interval. Since the batch jobs have different start times, the likelihood that their application-level checkpoints are written concurrently to the PFS is very small. Furthermore, we assume that each batch job allocates between 40%-90% of the memory available on each node. In this case, the size of the system-level checkpoint on each node coincides with the allocated memory. Out of this memory, we assume 20%-60% holds critical data structures that are written by application-level checkpointing approaches. This is the size of the application-level checkpoints. We use a random threshold for each batch job, both for the application-level and system-level checkpoints, which is then used in Equation 1 to calculate the application-level and system-level checkpointing duration.

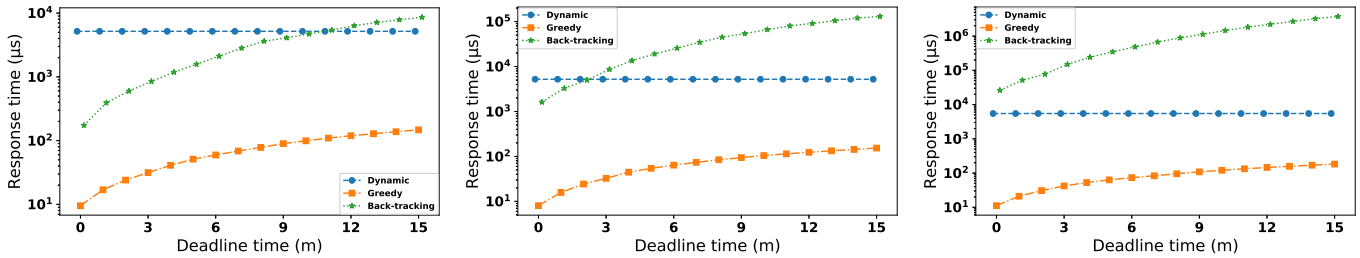
C. Compared approaches

Throughout our evaluations, we compare three approaches that can be used to solve the eviction problem:



(a) Evict at least K=512 nodes for on-demand jobs **(b)** Evict at least K=1024 nodes for on-demand jobs **(c)** Evict at least K=2048 nodes for on-demand jobs

Fig. 5: Response time for *Scenario-1* consisting of 12 batch jobs. Note the log scale on the Y axis. Lower is better.



(a) Evict at least K=512 nodes for on-demand jobs **(b)** Evict at least K=1024 nodes for on-demand jobs **(c)** Evict at least K=2048 nodes for on-demand jobs

Fig. 6: Response time for *Scenario-2* consisting of 16 batch jobs. Note the log scale on the Y axis. Lower is better.

1) Greedy: This algorithm implements a greedy strategy that tries to minimize the loss by checkpointing the most expensive jobs (high loss), while killing the least expensive jobs (low loss). To this end, it sorts the batch jobs in descending order of loss and tries to checkpoint them using the fastest available checkpointing method (application or system level). When the total checkpoint duration becomes larger than the deadline T, it iterates over the sorted jobs in reverse order starting from the end, killing them one by one until at least K nodes have been released. While it does not produce an optimal solution, this algorithm has linear complexity and therefore has a very fast response time.

2) Backtracking: This algorithm implements an exhaustive search of all possible choices for each batch job: keep running (exclude), kill, checkpoint at application-level, checkpoint at system-level. It optimizes the search by early abandoning of all combinations that cannot achieve a lower loss than the best combination found so far. Unlike *Greedy*, this approach always produces an optimal solution, however it has an exponential complexity and therefore may become untractable for large problem sizes.

3) CoSim: This is our proposal that implements Algorithm 1. It guarantees an optimal solution just like *Backtracking*, but at the same time it has a fast response time thanks to its polynomial complexity.

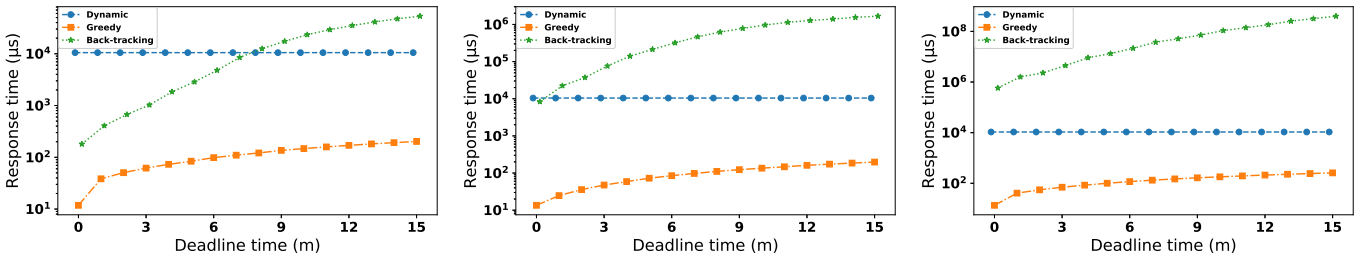
D. On-demand job configurations

For each of the three scenarios mentioned in Section IV-A, we consider the maximum deadline $T = 15$ minutes. We are interested in the optimal eviction strategy for all deadlines in the range $[0 \dots 15]$ with a granularity of one minute. Furthermore, for each of the three scenarios, we consider three different values for K, the minimum number of nodes that need to be evicted in order to make room for the on-demand jobs, i.e., 512, 1024, and 2048 for *Scenario-1*, *Scenario-2*, and *Scenario-3*, respectively. This roughly corresponds to 12.5%, 25% and 50% of the total capacity of Theta.

E. Results

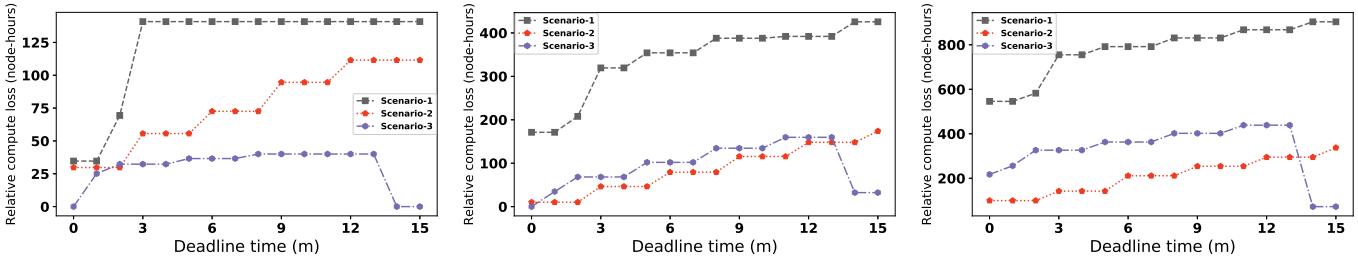
First, we focus on the performance and scalability of the three approaches. To this end, we measure the response time taken by each approach in order to produce the optimal eviction strategy for all deadlines in the range $[0 \dots T]$. As a consequence, in the case of *Greedy* and *Backtracking*, a separate run is executed for each $i \in [0 \dots T]$. Therefore, for an increasing i, the response time measures the accumulated runtime of all i runs. In the case of *CoSim*, a single run is sufficient to obtain the full solution thanks to the memoization of overlapping sub-problems. This metric is important because it determines how soon the scheduler can take decisions, which in turn impacts the value that can be extracted from the on-demand jobs (i.e., faster response time leads to better on-demand job results).

The results for each of the three scenarios are depicted in Figure 5, Figure 6 and Figure 7 respectively. Note that due to the large differences in algorithmic complexity between the three approaches, the y-axis is represented in a log scale. As expected, *CoSim* keeps a constant response time regardless of the deadline T. Despite the accumulation of response time from an increasing number of runs, the *Greedy* approach is still at least 30x faster than the other two approaches thanks to its linear complexity. It is interesting to observe that for a small K and a small number of batch jobs (as illustrated by *Scenario-1*), *Backtracking* is faster than our approach. However, with increasing K and number of batch jobs (as illustrated by *Scenario-2* and *Scenario-3*), the limitation of the exponential



(a) Evict at least K=512 nodes for on-demand jobs **(b)** Evict at least K=1024 nodes for on-demand jobs **(c)** Evict at least K=2048 nodes for on-demand jobs

Fig. 7: Response time for *Scenario-3* consisting of 24 batch jobs. Note the log scale on the Y axis. Lower is better.



(a) Evict at least K=512 nodes for on-demand jobs **(b)** Evict at least K=1024 nodes for on-demand jobs **(c)** Evict at least K=2048 nodes for on-demand jobs

Fig. 8: Relative compute loss of the *Greedy* approach relative to optimal solution produced by *CoSim* and *Backtracking*. Lower is better.

search becomes clearly visible, despite the aggressive early pruning optimization. In this case, our approach is up to five orders of magnitude faster. As a general conclusion, we observe that our approach has the advantage of providing the optimal solution within a predictable constant time, which is well suited for real-time scheduling decisions.

Next, we focus on the quality of the results of the *Greedy* approach. Since both our approach and the *Backtracking* approach produce the optimal solution, we use it as a baseline that we substract from the minimum loss found by the *Greedy* approach. We call this the relative compute loss. This metric is important, because it indicates what result quality degradation can be expected in order to benefit from faster response time.

As can be observed in Figure 8, the relative compute loss is very high, indicating that degradation in the quality of the result found by *Greedy* is unacceptable. In fact, with the exception of $T > 13$ for *Scenario-3*, the relative compute loss is increasing for an increasing T, which means *Greedy* suffers from an increasing degradation in the quality of the result. Also, it is important to note that in absolute terms, the minimum compute loss is decreasing with an increasing T for all three approaches, because more checkpointing opportunities become available. In fact, the optimal minimum loss is found by *CoSim* and *Backtracking* is often 0 (meaning no job needs to be killed), especially for larger T. Therefore, even when the relative compute loss seems to decrease for an increasing T, it is still missing the optimal compute loss by a large margin.

Based on this observation, we conclude that sacrificing the result quality for faster response time is not beneficial, especially when considering that our approach runs in the order of milliseconds in the worst case.

V. Related Work

Scheduling of batch and on-demand jobs for concurrent execution where resources sharing is limited to each type of job has been widely studied [12]–[19] in the past. However, not much work has been done for collocating both batch and on-demand jobs on the same set of resources [20]. SPRUCE (Special Priority and Urgent Computing Environment) [21] supports on-demand jobs by considering a basic preemptive scheduling scheme with no checkpointing. However, this leads to a significant loss of progress for the batch jobs.

Checkpointing based preemptive scheduling has been traditionally used at the operating system level for multi-tasking. However, recent checkpointing-based preemptive scheduling schemes [13], [22] focus on reducing their overheads and improving their effectiveness in reducing the average job turnaround time. Nevertheless, these techniques do not directly address the challenges of co-scheduling batch and on-demand jobs in HPC settings.

Large-scale datacenters operated by industry (e.g., Facebook [23] and Google [24]), leverage centralized job execution environments where the centralized system accumulates jobs from multiple datacenters, and then runs the computation [25]. However, it leads to increased network traffic and job completion time when the data volume grows exponentially [26], [27]. Furthermore, regulations may restrict moving data across continents due to security and privacy constraints, thus making such approaches impractical to adopt in production environments at large.

VI. Conclusions

In this paper, we present *CoSim*, a simulator that enables on-the-fly analysis of the trade-offs arising between delaying the

start of opportunistic on-demand jobs, which leads to longer analytics latency, and loss of progress due to preemption of batch jobs, which is necessary to make room for such on-demand jobs. The key idea of our proposal is to implement preemption through a combination of either killing or checkpointing (at application-level or system-level) a subset of batch jobs running on the compute nodes to free enough nodes by a given deadline. To this end, we introduce a checkpointing and loss model to develop a dynamic programming algorithm to minimize the loss for a variable deadline up to a given threshold, which gives the scheduler high flexibility in exploring a wide range of alternatives. *CoSim* finds the optimal solution up to 5 orders of magnitude faster than backtracking approaches and offers a predictable response time in the order of milliseconds, thereby eliminating the need for greedy approaches that are fast but find only approximate solutions.

In the future, we plan to investigate several avenues: (1) applicability of our proposal to cloud computing; (2) refinement of checkpointing model (interval, interactions with PFS); (3) integration with the workload schedulers at Argonne National Laboratory's supercomputers to validate *CoSim* for real-life on-demand workloads.

Acknowledgments

This material is based upon work supported by the U.S. Department of Energy (DOE), Office of Science, Office of Advanced Scientific Computing Research and Argonne National Laboratory. Results presented in this paper are obtained using the Chameleon and CloudLab testbeds supported by the National Science Foundation.

References

[1] W. Tang, B. Want, S. Ethier, and Z. Lin, "Performance portability of hpc discovery science software: Fusion energy turbulence simulations at extreme scale," *Supercomputing frontiers and innovations*, vol. 4, no. 1, 2017.

[2] A. S. Kozelkov, V. V. Kurulin, S. V. Lashkin, R. M. Shagaliev, and A. V. Yalozo, "Investigation of supercomputer capabilities for the scalable numerical simulation of computational fluid dynamics problems in industrial applications," *Computational Mathematics and Mathematical Physics*, vol. 56, no. 8, pp. 1506–1516, 2016.

[3] P. Vranas, G. Bhanot, M. Blumrich, D. Chen, A. Gara, P. Heidelberger, V. Salapura, and J. C. Sexton, "The bluegene/l supercomputer and quantum chromodynamics," in *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, Tampa, Florida, 2006, pp. 50–57.

[4] S. R. Ellingson, J. C. Smith, and J. Baudry, "Polypharmacology and supercomputer-based docking: opportunities and challenges," *Molecular Simulation*, vol. 40, no. 10-11, pp. 848–854, 2014.

[5] D. AOCNP, "Watson will see you now: a supercomputer to help clinicians make informed treatment decisions," *Clinical journal of oncology nursing*, vol. 19, no. 1, p. 31, 2015.

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

[7] N. Desai, "Cobalt: an open source platform for hpc system software research," in *Edinburgh BG/L System Software Workshop*, 2005, pp. 803–820.

[8] G. Staples, "Torque resource manager," in *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, New York, NY, USA, 2006, p. 8–es.

[9] J. Ansel, K. Arya, and G. Cooperman, "DMTCP: Transparent checkpointing for cluster computations and the desktop," in *IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, Rome, Italy, 2009, pp. 1–12.

[10] B. Nicolae, A. Moody, E. Gonsiorowski, K. Mohror, and F. Cappello, "Veloc: Towards high performance adaptive asynchronous checkpointing at large scale," in *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Rio de Janeiro, Brazil, 2019, pp. 911–920.

[11] J. Daly, "A higher order estimate of the optimum checkpoint interval for restart dumps," *Future Generation Computer Systems*, vol. 22, no. 3, pp. 303 – 312, 2006.

[12] R. Tyagi and S. K. Gupta, "A survey on scheduling algorithms for parallel and distributed systems," in *Silicon Photonics & High Performance Computing*. Singapore: Springer, 2018, pp. 51–64.

[13] V. J. Leung, G. Sabin, and P. Sadayappan, "Parallel job scheduling policies to improve fairness: A case study," in *International Conference on Parallel Processing Workshops (ICPP)*, San Diego, USA, 2010, pp. 346–353.

[14] A. A. Chandio, K. Bilal, N. Tziritas, Z. Yu, Q. Jiang, S. U. Khan, and C.-Z. Xu, "A comparative study on resource allocation and energy efficient job scheduling strategies in large-scale parallel computing systems," *Cluster computing*, vol. 17, no. 4, pp. 1349–1367, 2014.

[15] A. W. Mu'alem and D. G. Feitelson, "Utilization, predictability, workloads, and user runtime estimates in scheduling the ibm sp2 with backfilling," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 12, no. 6, pp. 529–543, 2001.

[16] C. Gómez-Martín, M. A. Vega-Rodríguez, and J.-L. González-Sánchez, "Fattened backfilling: An improved strategy for job scheduling in parallel systems," *Journal of Parallel and Distributed Computing (JPDC)*, vol. 97, pp. 69–77, 2016.

[17] B. Lawson and E. Smirni, "Multiple-queue backfilling scheduling with priorities and reservations for parallel systems," *ACM SIGMETRICS Performance Evaluation Review*, vol. 29, pp. 72–87, 2002.

[18] A. Tousimojarad and W. Vanderbauwhede, "An efficient thread mapping strategy for multiprogramming on manycore processors," *Parallel Computing: Accelerating Computational Science and Engineering (CSE), Advances in Parallel Computing*, vol. 25, pp. 63–71, 2014.

[19] S. G. Ahmad, C. S. Liew, M. M. Rafique, E. U. Munir, and S. U. Khan, "Data-intensive workflow optimization based on application task graph partitioning in heterogeneous computing systems," in *IEEE International Conference on Big Data and Cloud Computing (BdCloud)*, 2014, pp. 129–136.

[20] D. Wang, E.-S. Jung, R. Kettimuthu, I. Foster, D. J. Foran, and M. Parashar, "Supporting Real-Time Jobs on the IBM Blue Gene/Q: Simulation-Based Study," in *Job Scheduling Strategies for Parallel Processing*, D. Klusáček, W. Cirne, and N. Desai, Eds. Orlando, USA: Springer International Publishing, 2018, pp. 83–102.

[21] N. Trebon, "Enabling urgent computing within the existing distributed computing infrastructure," Ph.D. dissertation, University of Chicago, USA, 2011.

[22] Q. Snell, M. Clement, and D. Jackson, "Preemption based backfill," in *Job Scheduling Strategies for Parallel Processing*. Berlin, Heidelberg: Springer, 2002, pp. 24–37.

[23] J. Meza, T. Xu, K. Veeraraghavan, and O. Mutlu, "A large scale study of data center network reliability," in *ACM Internet Measurement Conference (IMC)*, New York, USA, 2018, p. 393–407.

[24] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat, "B4: Experience with a Globally Deployed Software Defined WAN," in *ACM SIGCOMM*, Hong Kong, China, 2013.

[25] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford, "Spanner: Google's globally distributed database," *ACM Transactions on Computer Systems (TOCS)*, vol. 31, no. 3, 2013.

[26] A. Vulimiri, C. Curino, P. B. Godfrey, T. Jungblut, J. Padhye, and G. Varghese, "Global analytics in the face of bandwidth and regulatory constraints," in *USENIX Networked Systems Design and Implementation (NSDI)*, USA, 2015, p. 323–336.

[27] S. Muralidhar, W. Lloyd, S. Roy, C. Hill, E. Lin, W. Liu, S. Pan, S. Shankar, V. Sivakumar, L. Tang *et al.*, "f4: Facebook's Warm BLOB Storage System," in *USENIX Operating Systems Design and Implementation (OSDI)*, 2014, pp. 383–398.