Evaluating the Potential of Coscheduling on High-Performance Computing Systems

Jason Hall[1], Arjun Lathi[2], David K. Lowenthal[3], and Tapasya Patki[4]

[1] iBoss, Inc.
[2] Expedia, Inc.
[3] Department of Computer Science, The University of Arizona
[4] Lawrence Livermore National Laboratory

Abstract. Modern high-performance computing (HPC) system designs have converged to heavyweight nodes with growing numbers of processors. If schedulers on these systems allocate nodes in an exclusive and dedicated manner, many HPC applications and scientific workflows will be unable to fully utilize and benefit from such hardware. This is because at such extreme scale, it will be difficult for modern HPC applications to utilize all of the node-level resources on these systems.
In this paper, we investigate the potential of moving away from dedicated node allocation and instead using *intelligent coscheduling*—where multiple jobs can share node-level resources—to improve node utilization and therefore job turnaround time. We design and implement a coscheduling simulator, and, using traces from a high-end HPC cluster with 100K jobs and 1158 nodes, demonstrate that coscheduling can improve average turnaround times by up to 18% when compared to easy backfilling. Our results indicate that coscheduling has the potential to be a more efficient way to schedule jobs on high-end machines in both turnaround time and system and component utilization.

Keywords: coscheduling · high-performance computing.

1 Introduction

Modern high-performance computing (HPC) system designs have converged to *heavyweight* nodes with growing numbers of CPUs that are supported by accelerators such as GPUs. Several top supercomputers [23], such as Frontier, Lumi, Leonardo, Sierra, and Summit, have tens of CPU cores and multiple GPUs on each node. These designs have many advantages, including better scaling [42], reduced power and network switch costs, and decreased network interference [36].

The heavyweight node trend should motivate the HPC community to rethink scheduling policies that allocate a set of dedicated nodes to each job. Dedicated node allocation is the most common policy on high-end systems and provides predictable performance and efficient execution when

nodes have modest resources. However, with modern systems, dedicated node allocation will often cause untenable levels of internal fragmentation of *multiple* node resources, including CPUs, GPUs, memory capacity, memory bandwidth, and network capacity—leading to significant underutilization of available resources.

Cloud installations, including scheduling infrastructures in Mesos [16], Kubernetes [31], and YARN [41], provide infrastructure for scheduling multiple jobs onto a node or nodes (to increase profit), but not in a way appropriate for HPC applications. Specifically, the HPC user base has more stringent performance expectations than cloud users and will not tolerate arbitrary performance degradation for their applications.

This paper studies the potential of *intelligent coscheduling* on modern HPC systems. We define intelligent coscheduing as scheduling multiple jobs, concurrently, onto overlapping nodes such that average job turnaround time is decreased and performance degradation for most jobs is at most modest. Such an approach accepts (some) intra-node interference between jobs rather than rigidly avoiding it via dedicated node allocations. Doing so will make a greater fraction of a system's resources available to jobs. Intelligent coscheduling will lower average turnaround time by decreasing the average time a job spends waiting to execute. This in turn will improve overall system utilization and throughput.

In particular, this paper is focused on studying and understanding the decrease in average turnaround time when using coscheduling compared to backfilling, which is generally the de-facto scheduling approach on high-end HPC installations. This paper focuses only on sharing nodes (and their memory) in multi-node applications. We describe our design and implementation of coscheduling and backfilling and provide results that show that coscheduling leads to lower average turnaround times: up to 18% compared to backfilling and over 80% compared to first-come, first-served. The downside is that some individual applications take longer to execute (once started), but the substantial decrease in wait time still leads to an average decrease in turnaround time.

The rest of this paper is organized as follows. Section 2 provides an overview of HPC scheduling and provides our assumptions made in this paper. Section 3 describes our implementation of backfilling and coscheduling. Section 4 explains our experimental setup, and Section 5 provides the experimental results. Finally, Section 6 discusses related work and Section 7 summarizes the paper.

2 Background and Assumptions

In this paper, we assume a high-performance computing system with a total of N nodes. Each node contains C cores. This section first reviews briefly the basics of HPC scheduling, and then proceeds to discuss our job submission assumptions and coscheduling.

2.1 Traditional HPC Job Scheduling

With traditional HPC job scheduling, each application submitted for execution on the system requests n nodes, where $1 \leq n \leq N$. The job also specifies an estimated runtime. When the scheduler has n nodes available and the job is at the front of the queue, the job is scheduled. The job is assigned all C cores, but actually uses c cores, where $1 \leq c \leq C$. (There may be a performance benefit to using fewer than C cores because of a decrease in memory contention.) The estimated runtime is used to set a deadline for the job, and the job is terminated if it exceeds this deadline.

Traditional HPC systems typically use dedicated node allocation (often called *space sharing* or *space slicing* [10]). Here, every node in the system is assigned to at most one job. This policy provides high system utilization, and, equally importantly, relatively predictable performance. In particular, it reserves all the memory on each of the n nodes for a single application. This avoids competition between multiple competing applications for memory, which can potentially cause thrashing (if the system is paged) or worse, a system crash in non-paged systems if the aggregate memory demand between the applications exceeds the size of physical memory.

2.2 Evaluating Scheduling Policies

There are many ways to evaluate different scheduling policies, including job throughput, system utilization, and average job turnaround time. Job throughput is defined as the number of jobs completed per unit time. System utilization is defined as the average fraction of nodes that are busy. Finally, the average job turnaround time across all jobs is simply the average wallclock time it takes from the time a job is submitted until the time it is completed.

It is well known that a typical first-come, first served (FCFS) scheduling policy will fall short in all three of these dimensions. If a job in an FCFS scheduling system cannot be scheduled because there are not enough nodes, the job *and* all jobs behind it on the queue are blocked, leading to a convoy effect.

2.3 Backfilling

The typical way HPC systems improve all of throughput, utilization, and turnaround time is to use *backfilling* [21, 24, 35, 18]. A backfilling policy addresses the convoy effect caused by FCFS by executing smaller jobs out of order on idle nodes so that utilization is improved—in turn reducing the overall average turnaround time. Backfilling is available in modern resource managers such as SLURM [2] or Flux [3].

However, backfilling is a strategy that is incentivized by utilization being defined as the number of busy nodes, rather than the number of busy *components* of nodes. With modern nodes containing large values of C along with on-board accelerators, this definition of utilization is outdated—a node can be busy, but its components may be ill-utilized. Accordingly, we look to an alternative scheme—coscheduling—to improve all three of throughput, utilization, and turnaround time. However, this requires relaxing job requirements.

2.4 Job Configuration Assumptions

In this work, we assume a more flexible job submission scheme. We assume that jobs are *malleable*, meaning they adapt to the number of nodes actually assigned to them [9]. Specifically, applications can be run with (i.e., assigned) *many* different values of n (nodes) and c (cores per node). We denote $(n \times c)$ as a *configuration*, and the set of all configurations in which the application can be executed as $\mathcal{S}$. The user submits a job with $\mathcal{S}$, which asserts to the job scheduler that the application can be executed with any configuration in $\mathcal{S}$. The user also specifies one of these configurations as the *base configuration*, denoted B.

We assume that configurations cannot have more total cores than the number used in B. As an example, assume that B for a job is $(n \times c)$. In this case, the user might include configurations $(2n \times c/2)$ and $(n/2 \times c)$ in $\mathcal{S}$. Configurations are not limited to ones with $n \times c$ cores; for example, we term the aforementioned $(n/2 \times c)$ as a *degraded* configuration because the number of total cores is less than that in the base configuration. The incentive for a user submitting a job with many configurations, including degraded ones, is potentially lower turnaround time via coscheduling. (Moreover, as discussed later, using configurations such as $(2n \times c/2)$ potentially benefit a job by decreasing memory pressure.)

3 Implementation

This section describes our implementations of backfilling and coscheduling.

3.1 Backfilling

Backfilling has two primary variants: *easy* and *conservative*. Easy backfilling allows short jobs to execute out of order as long as they do not delay the *first* queued job. Conservative backfilling, on the other hand, only allows short jobs move ahead if they do not delay *any* queued job. Easy backfilling performs better for most workloads [24]. In addition, backfilling algorithms frequently use a greedy algorithm that picks the *first-fit* from the set of available jobs in the job queue. The *first-fit* might not always be the *best-fit*, and a job further down the queue may fit the backfilled hole better. Finding the *best-fit* involves scanning the entire job queue, which increases job scheduling overhead significantly [34].

We use an Easy backfilling algorithm with first-fit. Our implementation skips over the first job at the head of the queue, in the case that (a) that job cannot be run given the currently available resources and has to wait until other jobs finish and (b) there exists a job that satisfies backfill constraints. We allow up to 150 jobs in the backfilling window, and when we backfill, the job at the head of the queue receives a reservation.

3.2 Coscheduling

As with FCFS, our coscheduling algorithm considers jobs in order on the submission queue. For each job at the head of the queue, our coscheduler either places the job on the system (and the job commences execution), or it blocks and no job can execute. The fundamental difference with our coscheduler is that a job can be placed on nodes that are already (partially) occupied by another job.

Coscheduling Benefits and Penalties There are a number of things to consider with a coscheduling implementation.

- A job that uses fewer cores per node will potentially achieve a speedup due to an increase in available memory bandwidth. We denote this as $memorySpeedup$.
- A job that is coscheduled on a node will potentially have a penalty due to a decrease in effective memory bandwidth. We denote this as $memorySlowdown$.

- A job that uses more (fewer) nodes will potentially have a remote communication penalty (benefit) due to an increase (decrease) in internode communication. We denote this as *communicationSlowdown*.
- A job that uses fewer cores than its base configuration will have a slowdown penalty. We denote this as *degradationSlowdown*.

We assume that we know (or can derive from known quantities) all of *memorySpeedup*, *memorySlowdown*, *communicationSlowdown*, and *degradationSlowdown*. Section 4 provides further details.

Coscheduling Implementation We found that with coscheduling, a first-fit approach produces poor results because it fails to (1) take advantage of the memory benefit when spreading a job out and (2) consider all configurations. Our coscheduler instead provides a modified best-fit approach.

Our modified best-fit coscheduling procedure is shown in Algorithm 1. For each job J at the head of the queue, we execute function `PlaceJob`. This function iterates through all possible configurations, and for each one we find the best placement. In this context, *best* means the placement that leads to the fastest execution for the job; i.e., we do not consider potential changes in execution time of other jobs sharing nodes with J. (Hence, our best-fit algorithm is greedy in the sense that it focuses only on J, and it also will allocate [completely] unoccupied nodes if possible. Another option, which we leave for future work, is to allow a job to specify a maximum slowdown it is willing to incur; such an approach has been used in other contexts [28].)

Function `GetBestPlacement` finds the best placement for a set of nodes and a given configuration used by J. First, we calculate the potential speedup due to memory effects (function `GetMemoryBenefit`) for the given configuration. Then, for each node in the system that is usable (determined by function `CanSchedule`), we determine the memory slowdown (and therefore the relative node performance) for J. To make this determination, we take into account the memory slowdown that occurs due to other jobs already executing on the node; the memory sensitivity of J is used in this calculation. A node is usable if it has a sufficient number of free cores to accommodate the configuration. (Optionally, our implementation allows for hard limits on the number of jobs that can be coscheduled on a node.)

Function `GetBestPlacement` adds the node to the allocation for J if either (1) we do not yet have the target number of nodes needed by the configuration, or (2) the slowdown for J will decrease if the node replaces

Algorithm 1 Sketch of coscheduling algorithm.

```
function PlaceJob(job)
    for config in Configurations do
        bestPlacement = None
        bestRunTime = Infinity
        (placement, relNodePerf) = GetBestPlacement(job, config)
        runTime = EstimateRunTime(config, relNodePerf)
        if runTime < bestTime then
            bestRunTime = runTime
            bestPlacement = placement
    return bestPlacement

function GetBestPlacement(job, config)
    memSpeedup = GetMemoryBenefit(config)
    allocatedNodes = {}
    for node in Nodes do
        if CanSchedule(node) then
            memSlowdown = DetermineMemorySlowdown(job.sensitivity, node)
            #relNodePerf is the relative node performance; can be faster or slower,
            #depending on which of memSpeedup and memSlowdown is larger
            relNodePerf = GetNodeSlowdown(memSpeedup, memSlowdown, node)
            if len(allocatedNodes) < config.nodes then
                allocatedNodes.insertSorted(node, relNodePerf)
            else if tail(allocatedNodes).relNodePerf > relNodePerf then
                allocatedNodes.remove(tail(allocatedNodes))
                allocatedNodes.insertSorted(node, relNodePerf)
    if len(allocatedNodes) == config.nodes then
        return (allocatedNodes, tail(allocatedNodes).relNodePerf)
    else
        return ([], -1)

function DetermineMemorySlowdown(sensitivity, node)
    for each other job J' on this node do
        determine and add in slowdown due to effect on J by J'
    return memSlowdown

function EstimateRunTime(config, relNodePerf)
    if config.isDegraded then
        computationTime = config.baseRunTime * degradationSlowdown
    computationTime = computationTime * relNodePerf
    communicationTime = config.communicationTime * communicationSlowdown
    runTime = computationTime + communicationTime
    return runTime

function GetMemoryBenefit(config)
    fractionCores = config.coresPerNode / systemCoresPerNode
    return improvement[fractionCores]

function CanSchedule(node)
    return node.freeCores >= config.numCores and within coscheduling limit
```

a node that currently is part of the allocation. Note that the slowdown for the job is the maximum slowdown over all nodes in J.

Function `PlaceJob` iterates through all configurations (and all nodes in each configuration)[5]. If an allocation is found with a sufficient number of nodes, then the runtime (relative to the base configuration) of J is computed. If the runtime is better than the best known runtime, this placement is marked as the current best. As some configurations use more nodes and fewer cores/node, jobs have the potential to have *lower* execution time than would be possible in a strict FCFS (or backfilled) scheduler.

Our implementation has hooks to allow several variations within our modified best-fit algorithm. The variations are all based on restricting what jobs can be coscheduled on the same nodes. One such variation is preventing two jobs that have high memory sensitivity from executing on the same node, which avoids a large penalty due to memory interference. Section 5 provides results of using these variations.

4 Experimental Setup

We next describe our experimental infrastructure and evaluation setup. Our simulator is written in python and allows us to run experiments with different cluster sizes and numbers of jobs. The simulator takes as input a trace that contains, for each job, an entry that contains a unique id, arrival time, execution time, and number of nodes used. It also contains the coscheduling benefits and penalties described in the previous section. The way the benefits and penalties are chosen is discussed below.

The particular trace used is from Cab [12], a (now decommissioned) cluster at Lawrence Livermore National Laboratory that had 1296 nodes, but commonly operated with 1158 nodes in the batch partition. While the Cab trace has nearly 700,000 jobs, to keep simulation time manageable we used subsets of 100,000 jobs taken from this trace for each experiment. The first job was selected after job number 50,000 to avoid any anomalies having to do with machine bootup; we experimented with four traces in all, each having a unique, non-overlapping set of jobs.

The simulator can handle arbitrary numbers of nodes and cores per node, but as Cab had 1158 nodes and 16 cores per node, we used this configuration for all experiments except the one that studies varying the

[5] This paper is focused on studying the viability of coscheduling, so we trade off scheduling time (traversing all nodes) for potentially better decisions. There are many ways to improve search time, including ordering the nodes by free cores.

number of nodes. (Other Cab features, such as a Sandy Bridge node architecture, are irrelevant for our study.) The base configuration for a job is therefore $(n \times 16)$. For our backfilling experiments, we used EASY backfilling with a window size of 150.

While our simulator is not optimized for speed of simulation, it is important to ensure that coscheduling does not take an unreasonably long time to place jobs. The average time for placing a job is less than 5ms for coscheduling, 4ms for easy backfilling, and 400us for FCFS. Notably, coscheduling and backfilling are reasonably comparable, and both are, as expected, much slower than FCFS.

For coscheduling experiments, it is necessary to add job-level information. The coscheduling benefits and penalties are chosen from a uniform random distribution with the following endpoints:

- Memory sensitivity: not sensitive (denoted "low"), somewhat sensitive (denoted "moderate"), or very sensitive (denoted "high");
- Percentage of total time spent in communication: chosen between 1% and 20%;
- Communication penalty: chosen between 0% and 40% (applied only to the communication portion); and,
- Degradation penalty: chosen between a slowdown of 1.5 and 2 each time the number of cores is halved.

Each of these (randomized) values is added to the entry of each job in the trace. In general it would be necessary to automatically generate this information; for example, Tang et. al [38] used a database to store application performance data that could be used if the job were executed again. While we envision a different approach to gathering such information, as this paper is a study to determine the potential extent of the benefits of coscheduling, we assume that this information is known and stored in the job file on a per-job basis.

We note that determining memory sensitivity in a clear and concise form in general is a challenging problem. This paper is focused on evaluating the potential of coscheduling. There are a number of ways to estimate memory sensitivity, including the aforementioned keeping of a database of prior program executions [38], executing a so-called skeleton version of the code to infer behavior, and taking input from the user through program annotations or other interface. (The last approach has the potential problem of dishonest users.) However, handling discovery and representation of memory sensitivity of programs falls outside the scope of this paper, and we leave this for future work.

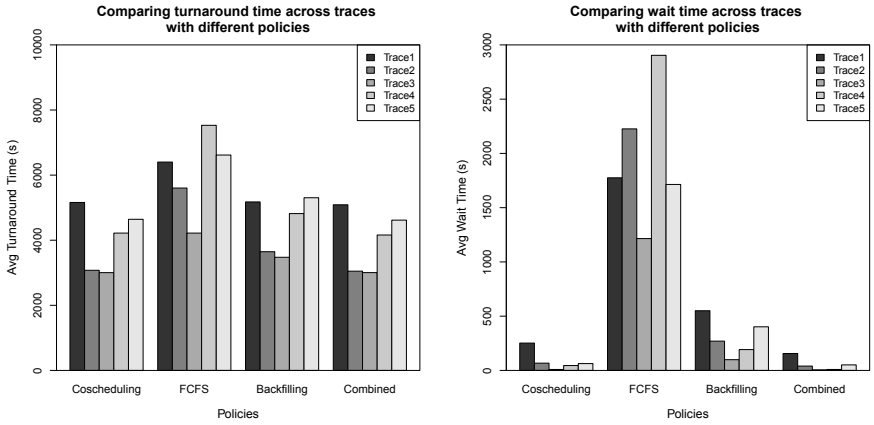

Fig. 1. Comparing Coscheduling, FCFS, Backfilling, and Coscheduling with Backfilling ("Combined") on Trace1 through Trace5. The left graph shows turnaround time, and the right graph shows wait time. For readability, the y-axis ranges are different in the two graphs.

Given that we are assuming 16-core nodes, our prototype considers these configurations for every job: $(4n \times 4)$, $(2n \times 8)$, $(n \times 16)$, $(2n \times 4)$, $(n \times 8)$, $(n/2 \times 16)$, $(n/2 \times 8)$, where the base configuration is $(n \times 16)$, and the last four are degraded configurations as they use less than $n \times 16$ cores. We ran experiments with additional configurations (e.g., $(8n \times 2)$), but they did not perform as well. Similarly, we set a maximum number of coscheduled jobs per node of three; with the configurations considered, the hard limit is four, but we found that led to inferior results.

5 Results

This section presents our results. We begin by discussing turnaround times for coscheduling, backfilling, and FCFS. Next, we discuss the impact of coscheduling on job execution times. The following two subsections discuss varying the number of nodes and coscheduling restrictions. Finally, we provide a short discussion of the implications of our results.

5.1 Turnaround Time Results

Average Turnaround Time Results Figure 1 (left) shows average turnaround times for coscheduling, FCFS, FCFS with easy backfilling (hereafter denoted as just "backfilling"), and coscheduling with backfilling.

These tests are conducted using different 100K sections (and so 100K jobs) of the Cab trace file (denoted Trace1, Trace2, Trace3, Trace4, and Trace5). When using coscheduling, for each trace, we ran five experiments, and each used a different random seed; the result shown is the one with the median average turnaround time. For our traces, the coefficient of variation was as low as 0.15% and as high as 1.09%. When using FCFS or backfilling, there is no randomness, so we only ran one experiment per trace.

Coscheduling results in turnaround times up to 18% lower than backfilling and over 80% lower than FCFS, due to smaller average wait times (shown in the right part of the figure). Not surprisingly, the performance is trace dependent. Coscheduling has lower average turnaround time than backfilling for all five traces (by between 1% [Trace1] and the aforementioned 18% [Trace2]), with four of the five traces showing a difference of 14% or more. The reason for the smaller gain on Trace1 is in part because of larger average wait time, which in turn is likely an attribute of the job mix in Trace1. This shows that scheduling algorithm performance is at least somewhat dependent on the characteristics of the job mix.

The figure also shows that coscheduling gains little additional benefit from backfilling; in our five traces, the improvement from adding backfilling to coscheduling is always less than 3%. This is not surprising, given that coscheduling is aggressively scheduling jobs onto partially occupied nodes, specifically to lower wait time—so, the wait time is *already* low.

Impact on Individual Job Turnaround Times The above results establish that coscheduling decreases *average* turnaround time compared to backfilling and FCFS. This subsection explores the effect of coscheduling on *individual* jobs; i.e., does the decrease in average turnaround time come at the cost of greatly increasing the turnaround time of specific jobs? (All results from here to the end of Section 5 use Trace1, because this is the trace where backfilling was most competitive with coscheduling.)

In Figure 2, we show three comparisons across the 100K jobs from Trace1. The vertical axis on these graphs represents the ratio of turnaround times of the policies being compared, in a manner that allows the reader to visualize when a certain policy results in better turnaround times than another. We begin by taking the ratio of turnaround times between two policies for each job. If this ratio is greater than one, we know that the second policy had a better overall turnaround time than the first policy. We plot these values as-is. If the ratio is less than one, we know that the first policy had a better overall turnaround time on the job, and we negate and invert this ratio for better readability. As a result, in the subgraphs

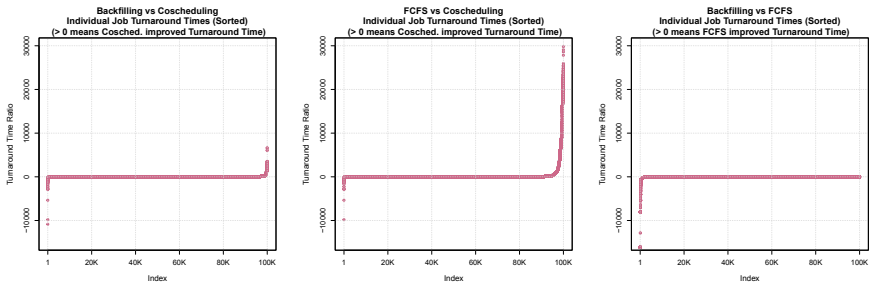

Fig. 2. Job Turnaround Time comparison across the three policies for Trace1.

shown in Figure 2, a value less than zero shows that the first policy had better turnaround time for a certain job and by what factor. Similarly, a value greater than zero indicates that the second policy did better and shows its associated improvement.

The first subgraph in Figure 2 compares coscheduling to backfilling. The key point is that at the extreme right on the x-axis, turnaround times for coscheduling are much better (over 5000x) than backfilling; the opposite is true at the extreme left on the x-axis. These extreme points are dominated by wait time; both coscheduling and backfilling encounter situations where the cluster is full or nearly full and a job incurs significant wait time—coscheduling just has that occur for different jobs than for backfilling. Given that which jobs end up incurring the large wait times is arbitrary for each of coscheduling and backfilling, coscheduling achieves its decrease in average turnaround time *without* significantly increasing worst-case wait times for individual jobs.

The second subgraph in Figure 2 shows the same job-level comparison for coscheduling compared to FCFS. The shape of the graph is similar, but as expected, there are many more large wait times for FCFS because of the convoy effect. Accordingly, coscheduling is better in average turnaround time and has fewer large wait times. For comparison, the third subgraph shows the job-level comparison for backfilling and FCFS. As can be seen, backfilling does not have any large job wait times compared to FCFS—in particular, no job in FCFS is more than 89% slower than backfilling. This does not mean than backfilling does not have jobs with large wait times—but rather that those jobs have even larger wait times with FCFS.

Because the data points in the middle are not easily readable on these subgraphs (due to the scale), Table 1 depicts the quartile data for the distributions. Note that the negative values in the table indicate the negation and inversion explained earlier.

Table 1. Quartiles of Turnaround Time Ratios for Policies

Policies	0%	25%	50%	75%	100%
Backfilling vs FCFS	-16187	-1.0900	1.0000	1.0000	1.8900
FCFS vs Coscheduling	-9799.0	-1.0080	1.0900	1.6800	29805
Backfilling vs Coscheduling	-10856	-1.0200	1.0500	1.3400	6631.6

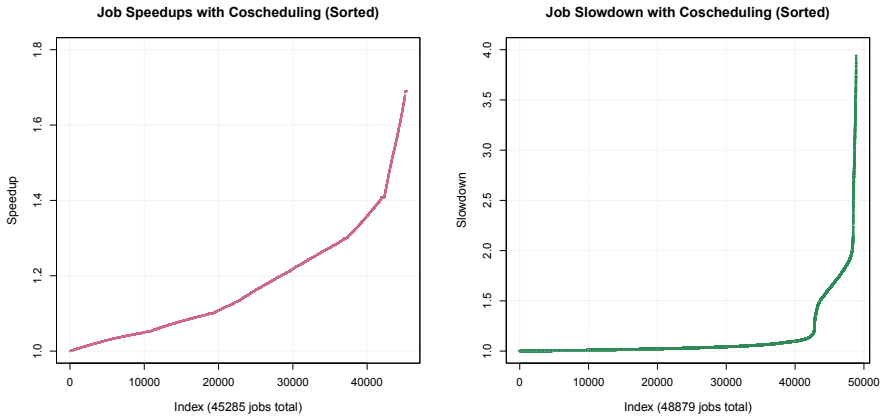

Fig. 3. Job Execution Time Speedup and Slowdown distributions (sorted) from Trace1

5.2 Impact on Individual Job Execution Times

This subsection explores the effect of coscheduling on individual job *execution* times. Figure 3 shows this effect. Depending on the configuration that an incoming job is assigned at allocation time and the other jobs it shares its allocation with, the incoming job may experience a speedup in execution time (due to reduced memory contention), no change in execution time, or a slowdown in execution time (due to decreased memory bandwidth, communication penalty, or degraded configuration, as described in Section 3.2). Note that the execution time of a job only experiences a change with the coscheduling policy—both FCFS and backfilling use the original execution time from the job trace and do not modify individual job execution times.

We analyzed one of the traces of 100K jobs (Trace1), where 45,285 jobs had decreased execution time (speedup), 48,879 jobs had increased execution time (slowdown), and 5,836 jobs had no change in execution time. The left graph in Figure 3 shows the sorted distribution of jobs that sped up, and the right graph shows the sorted distribution of jobs that slowed down. We observed that for the 48.8% jobs that slowed down, most

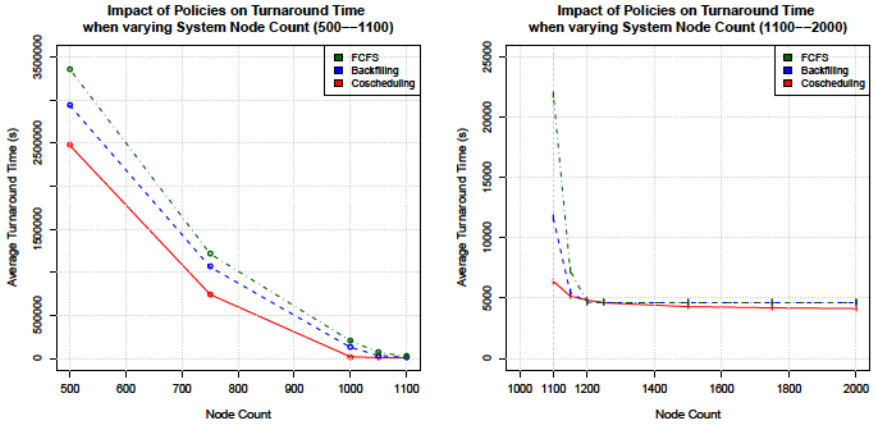

Fig. 4. Varying node count. Because of the large range of average turnaround times, two graphs are shown with different ranges of node count for better readability. Results use Trace1.

did not experience a significant slowdown. Only a little over 6% of the jobs slowed down by more than 20% in Trace1. Additionally, with coscheduling, the number of jobs that slowed down by factors of two, three, and four were 588, 226, and one, respectively. No job slowed down by more than a factor of five.

5.3 Differing Numbers of Nodes

Figure 4 shows the results of coscheduling, FCFS, and backfilling as the system node count varies from 500 to 2000. Two graphs are used for readability; please note that the x-axis of the left graph starts at 500 and goes to 1100, and the x-axis of the right graph starts at 1100 and goes to 2000. As with the previous results, coscheduling consistently leads to the lowest average turnaround time. The job trace we use is for a system of 1158 nodes, so when we use relatively small system node counts (e.g., 500), the average turnaround time is extremely high (as expected). Coscheduling leads to the best average turnaround time because wait time dominates this scenario (and coscheduling is designed to decrease wait time at the expense of execution time). When the node count is sufficiently large (1400 and above), all algorithms converge to a single value because there are enough nodes to avoid any waiting (again, the trace is for a system of 1158 nodes). Coscheduling converges to a lower average turnaround time because it leverages the additional nodes to spread out a larger number of jobs, achieving a memory bandwidth benefit on such jobs.

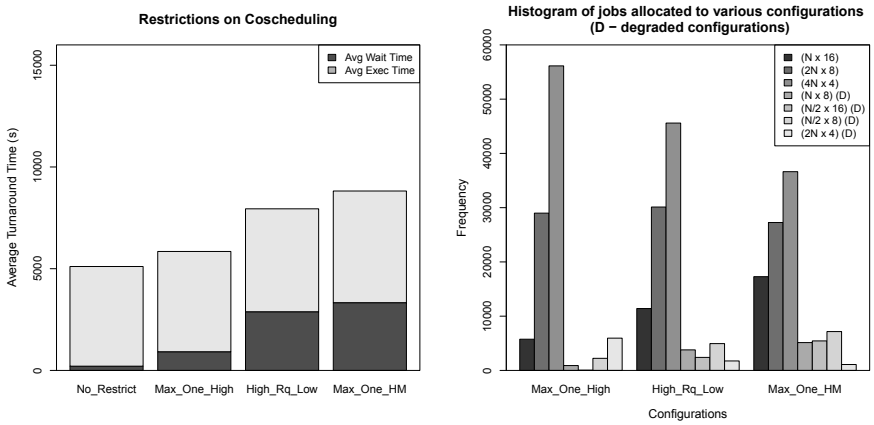

Fig. 5. Restricted coscheduling. The left graph shows four scenarios, from no restrictions (`No_Restrict`) to allowing only one high *or* medium sensitivity job on a node (`Max_One_HM`). The middle two bars are, in turn, allowing at most one high sensitivity job on a node (`Max_One_High`) and any high sensitivity job on a node requiring all other jobs on the node to be low sensitivity (`High_Rq_Low`). The right graph shows the distribution of configurations for the experiment. Results use Trace1.

5.4 Restricting Coscheduling

Our coscheduling algorithm places no restrictions on which types of jobs (low, moderate, high) can be coscheduled. An alternative coscheduling design attempts to limit coscheduling overhead due to memory contention by restricting coscheduling possibilities. Figure 5 shows the effect of doing so on Trace1.

In the left graph, each bar shows turnaround time, broken down into wait time and execution time. The leftmost bars show results of our base coscheduling algorithm (`No_Restrict`), which has no restrictions. Proceeding rightward, the next bar shows results if we restrict coscheduling to at most one high sensitivity job (`Max_One_High`). The third bar further restricts coscheduling to require a high sensitivity job to be coscheduled only with low sensitivity jobs (`High_Rq_Low`). Finally, the fourth bar requires that coscheduling occur with at most one high *or* moderate sensitivity job (`Max_One_HM`).

The figure shows that restrictions lead to higher turnaround times. Unsurprisingly, placing restrictions on which types of jobs can be coscheduled leads to higher wait times. Counterintuitively, restrictions also lead to higher execution times. The reason for this (see the right graph in the figure) is that the higher wait times cause degraded configurations to be

Table 2. Number of jobs blocked, due to fragmentation and capacity, with and without degraded configurations.

Version	Num blocked (fragmentation)	Num blocked (capacity)
With degraded configs	5,805	7,595
Without degraded configs	41,664	25,671

used more often (to avoid increasing wait times even more). The number of degraded configurations with `No_Restrict` is 6,106; in turn, that number is 9,121 for `Max_One_High`, 12,880 for `High_Rq_Low`, and 18,825 for `Max_One_HM`. Obviously, if average wait and average execution time both increase, so does average turnaround time. Fundamentally, one is generally better off just using FCFS than placing restrictions on coscheduling.

Finally, Table 2 shows the effect, specifically, of degraded configurations with coscheduling *without* restrictions. The table makes clear that degraded configurations are critical to the success of coscheduling as they avoid significant wait time. This is true both in terms of avoiding blocks due to fragmentation (there are sufficient total resources, but not in usable form) and capacity (there are insufficient total resources). Avoiding wait time in exchange for slightly slower execution time is a tradeoff that is clearly better.

6 Related Work

Allocating dedicated nodes has a long history at HPC centers because it prevents inter-job interference (especially due to cache/memory) on a node. However, it does not eliminate sharing the interconnect, and it does not prevent CPU interrupts on a node. CPU interrupts can be quite damaging to performance [29, 40]. For example, if a noise event occurs on just one out of N cores running identical code, for large N, the next global synchronization point will suffer a delay equal to the duration of the noise event [29, 14, 7]. Even if synchronization is local, this can still result in so-called cascading effects [17, 11]. The community has responded with many techniques to avoid problems due to core context switching. These include core specialization [2], lightweight operating systems [19], and zero-noise operating systems [39, 1].

Some of the work on coscheduling comes from the HPC scheduling community. For example, paired gang scheduling [44] had the goal of modifying gang scheduling to execute two jobs on a node, one CPU bound and the other I/O bound. In addition, virtualization often leads to consolida-

tion of applications (often done to conserve power), which is a form of coscheduling [43].

A significant amount of work on coscheduling comes from cloud installations, where the primary goal is the profit of the cloud provider. Kubernetes [31] and YARN [41] provide infrastructure for scheduling multiple jobs onto a node or nodes; application degradation is only limited by the service-level agreement. HPC schedulers for these frameworks only schedule individual applications to Cloud nodes [33], and cloud schedulers that gang-schedule processes generally implement only simple scheduling algorithms [27]. None of these approaches improves system-wide utilization and turnaround time for multiple, diverse HPC applications.

Two recent related papers have used coscheduling. Tang et al. [38] use coscheduling that results in jobs being spread out, similar to our technique. This work fixes the number of cores, whereas our approach considered degraded configurations. Saba et al. [32] use coscheduling to tackle the specific problem of CPU-GPU applications. They formulate an optimization problem and use machine learning to do the partitioning. Theoretical aspects of coscheduling for HPC with a focus on memory and resilience have also been explored in Pottier's dissertation [30].

Modern batch schedulers for HPC systems such as SLURM [2] and FLUX [3] support the assignment of multiple jobs to a node in the sense that one *could* use them to coschedule. However, they typically allocate dedicated nodes in practice to avoid performance penalties that can arise from sharing caches, main memory, network interfaces, or any other node-level resource [8, 15, 22, 25]. In addition, there is an inter-node penalty when multiple jobs share a node in that more pressure may be put on network links, resulting in network interference. Other scheduling policies other than allocating dedicated nodes exist, such as time-sharing, gang scheduling [13], or implicit coscheduling [5]. Gang scheduling and implicit coscheduling are ways to try to get all nodes of a job active at the same time, without resorting to space sharing. Using gang scheduling with admission control to avoid paging and thrashing has also been studied [6].

7 Summary

This paper has presented a coscheduling approach on a high-performance computing system and evaluated its effectiveness compared to FCFS and backfilling. While coscheduling may increase execution time, its decrease in wait time generally makes it a superior approach in situations where

average job turnaround time is most important. Combining backfilling and coscheduling can further reduce wait times, albeit marginally.

There are still many aspects of this problem to be studied. For example, we assumed we knew job characteristics, such as memory sensitivity or communication patterns, but in general such information must be discovered through historical data analysis of job traces or through performance counter based models. Similarly, many HPC systems include multiple GPUs per node, and application kernels can share a GPU [37, 20]. We did not study the impact of determining which application kernels work well together in a shared GPU environment, or how AI/ML workflows with GPUs would fare. We focused on average turnaround times and hence, job throughput, and we did not address system or component utilization. Modeling component utilization (such as cores, GPUs, memory) or flow resource utilization (such as power, network, or I/O bandwidth)—as well as identifying what occupancy or idleness means for each of these resources—is crucial to conduct a tradeoff analysis of utilization versus throughput when comparing backfilling and coscheduling. For GPUs, occupancy and utilization metrics are provided through vendor interfaces, such as the NVML or ROCm libraries [26, 4]. Designing a model to determine core occupancy or memory occupancy, or power utilization, however, is an area of future research. These are just a few of the avenues we will pursue, given the results here that coscheduling—applied judiciously—can improve the efficiency of HPC systems.

8 Acknowledgments

This work was performed in part under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-CONF-844869-DRAFT). In addition, this material is based upon work supported by the National Science Foundation under Grant No. 2103511. This research used resources of the National Energy Research Scientific Computing Center, a DOE Office of Science User Facility supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231. We would also like to thank the anonymous reviewers for their helpful suggestions.

References

1. Mckernel, https://www.pccluster.org/en/mckernel/download.html
2. Slurm workload manager, https://slurm.schedmd.com/core_spec.html
3. Ahn, D.H., Garlick, J., Grondona, M., Lipari, D., Springmeyer, B., Schulz, M.: Flux: A next-generation resource management framework for large HPC centers. In: Workshop on Scheduling and Resource Management for Parallel and Distributed Systems (September 2014)
4. AMD: AMD ROCm Platform Reference, https://cgmb-rocm-docs.readthedocs.io/en/latest/index.html
5. Arpaci-Dusseau, A.C.: Implicit coscheduling: Coordinated scheduling with implicit information in distributed systems. ACM Transactions on Computer Systems **19**(3), 283–331 (2001)
6. Batat, A., Feitelson, D.: Gang scheduling with memory considerations. In: International Parallel and Distributed Processing Symposium. pp. 109–114 (2000)
7. Beckman, P., Iskra, K., Yoshii, K., Coghlan, S.: The influence of operating systems on the performance of collective operations at extreme scale. In: International Conference on Cluster Computing (Sep 2006)
8. Delimitrou, C., Kozyrakis, C.: Paragon: Qos-aware scheduling for heterogeneous datacenters. SIGPLAN Not. **48**(4), 77–88 (Mar 2013)
9. Fan, Y., Lan, Z., Rich, P., Allcock, W.E., Papka, M.E.: Hybrid workload scheduling on HPC systems. In: IPDPS (2022)
10. Feitelson, D.G., Rudolph, L.: Parallel job scheduling: Issues and approaches. In: Workshop on Job Scheduling Strategies for Parallel Processing (1995)
11. Ferreira, K.B., Bridges, P., Brightwell, R.: Characterizing application sensitivity to OS interference using kernel-level noise injection. In: Supercomputing (Nov 2008)
12. Flux Framework Team: 2014 Cab supercomputer job scheduling traces (2020)
13. Franke, H., Pattnaik, P., Rudolph, L.: Gang scheduling for highly efficient, distributed multiprocessor systems. In: Proceedings of the 6th Symposium on the Frontiers of Massively Parallel Computation (1996)
14. Garg, R., De, P.: Impact of noise on scaling of collectives: An empirical evaluation. In: International Conference on High Performance Computing (Dec 2006)
15. Govindan, S., Liu, J., Kansal, A., Sivasubramaniam, A.: Cuanta: Quantifying effects of shared on-chip resource interference for consolidated virtual machines. In: ACM Symposium on Cloud Computing (2011)
16. Hindman, B., et al.: Mesos: A platform for fine-grained resource sharing in the data center. In: Networked Systems Design and Implementation (2011)
17. Hoefler, T., Schneider, T., Lumsdaine, A.: Characterizing the Influence of System Noise on Large-Scale Applications by Simulation. In: Supercomputing (Nov 2010)
18. Jackson, D., Snell, Q., Clement, M.: Core Algorithms of the Maui Scheduler. In: Job Scheduling Strategies for Parallel Processing (2001)
19. Lange, J., et al.: Palacios and Kitten: New high performance operating systems for scalable virtualized and native supercomputing. In: Proceedings of the 24th IEEE International Parallel and Distributed Processing Symposium (April 2010)
20. Li, B., Patel, T., Samsi, S., Gadepally, V., Tiwari, D.: Miso: Exploiting multi-instance gpu capability on multi-tenant gpu clusters. In: Proceedings of the 13th Symposium on Cloud Computing (2022)
21. Lifka, D.: The ANL/IBM SP Scheduling System. In: Job Scheduling Strategies for Parallel Processing, vol. 949, pp. 295–303 (1995)

22. Mars, J., Tang, L., Hundt, R., Skadron, K., Soffa, M.L.: Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations. In: IEEE/ACM International Symposium on Microarchitecture (2011)
23. Meuer, H., Strohmaier, E., Dongarra, J., Simon, H.: "Top500 Supercomputer Sites" (2022), http://www.top500.org
24. Mu'alem, A.W., Feitelson, D.: Utilization, Predictability, Workloads, and User Runtime Estimates in Scheduling the IBM SP2 with Backfilling. Parallel and Distributed Systems, IEEE Transactions on **12**(6), 529–543 (2001)
25. Nathuji, R., Kansal, A., Ghaffarkhah, A.: Q-clouds: Managing performance interference effects for qos-aware clouds. In: EuroSys (2010)
26. NVIDIA Corp.: NVML API Reference Guide, https://docs.nvidia.com/deploy/nvml-api/index.html
27. Palantir, I.: Spark scheduling in kubernetes (May 2019), https://medium.com/palantir/spark-scheduling-in-kubernetes-4976333235f3
28. Patki, T., Sasidharan, A., Melarth, M., Lowenthal, D.K., Rountree, B., Schulz, M., de Supinski, B.: Practical Resource Management in Power-Constrained, High Performance Computing. In: High-Performance Distributed Computing (Jun 2015)
29. Petrini, F., Kerbyson, D.J., Pakin, S.: The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of ASCI Q. In: Supercomputing (2003)
30. Pottier, L.: Co-scheduling for large-scale applications : memory and resilience. Ph.D. thesis, Université de Lyon (October 2018), https://theses.hal.science/tel-01892395/
31. Rensin, D.K.: Kubernetes - Scheduling the Future at Cloud Scale. 1005 Gravenstein Highway North Sebastopol, CA 95472 (2015), http://www.oreilly.com/webops-perf/free/kubernetes.csp
32. Saba, I., Arima, E., Liu, D., Schulz, M.: Orchestrated co-scheduling, resource partitioning, and power capping on CPU-GPU heterogeneous systems via machine learning. In: Architecture of Computing Systems (2022)
33. Saha, P., Beltre, A., Govindaraju, M.: Scylla: A Mesos framework for container based MPI jobs. CoRR **abs/1905.08386** (2019), http://arxiv.org/abs/1905.08386
34. Shmueli, E., Feitelson, D.: Backfilling with Lookahead to Optimize the Performance of Parallel Job Scheduling. In: Job Scheduling Strategies for Parallel Processing, Lecture Notes in Computer Science, vol. 2862 (2003)
35. Skovira, J., Chan, W., Zhou, H., Lifka, D.: The EASY LoadLeveler API Project. In: Job Scheduling Strategies for Parallel Processing, Lecture Notes in Computer Science, vol. 1162, pp. 41–47. Springer Berlin Heidelberg (1996)
36. Smith, S.A., Cromey, C.E., Lowenthal, D.K., Domke, J., Jain, N., Thiagarajan, J.J., Bhatele, A.: Mitigating inter-job interference using adaptive flow-aware routing. In: Supercomputing (Nov 2018)
37. Steve Rennich: CUDA C/C++ Streams and Concurrency, https://developer.download.nvidia.com/CUDA/training/StreamsAndConcurrencyWebinar.pdf
38. Tang, X., Wang, H., Ma, X., El-Sayed, N., Zhai, J., Chen, W., Aboulnaga, A.: Spread-n-share: Improving application performance and cluster throughput with resource-aware job placement. In: Supercomputing (2019)
39. The BlueGene/L Team: An overview of the BlueGene/L supercomputer. In: Proceedings of the 2002 ACM/IEEE Conference on Supercomputing (SC'02) (2002)
40. Tsafrir, D., Etsion, Y., Feitelson, D., Kirkpatrick, S.: System noise, os clock ticks, and fine-grained parallel applications. In: International Conference on Supercomputing (Jun 2005)

41. Vavilapalli, V.K., et al.: Apache hadoop yarn: Yet another resource negotiator. In: Proceedings of the 4th Annual Symposium on Cloud Computing (2013)
42. Vazhkudai, S.S., et al.: The Design, Deployment, and Evaluation of the CORAL Pre-Exascale Systems. In: Supercomputing (November 2018)
43. Verma, A., Ahuja, P., Neogi, A.: Power-aware dynamic placement of HPC applications. In: International Conference on Supercomputing (2008)
44. Wiseman, Y., Feitelson, D.: Paired gang scheduling. IEEE Transactions on Parallel and Distributed Systems **14**(6), 581–592 (2003)