

Job Coscheduling on Coupled High-End Computing Systems

Wei Tang,^{*} Narayan Desai,[†] Venkatram Vishwanath,[†] Daniel Buettner,[‡] Zhiling Lan^{*}

^{*}*Department of Computer Science, Illinois Institute of Technology
Chicago, IL 60616, USA*

wtang6@iit.edu, lan@iit.edu

[†]*Mathematics and Computer Science Division*

[‡]*Argonne Leadership Computing Facility*

^{†‡}*Argonne National Laboratory, Argonne, IL 60439, USA*

desai@mcs.anl.gov, venkatv@mcs.anl.gov, buettner@alcf.anl.gov

Abstract—Supercomputer centers often deploy large-scale computing systems together with an associated data analysis or visualization system. In this paper, we propose a coscheduling mechanism, providing the ability to coordinate execution between jobs on different systems. The mechanism is built on top of a lightweight protocol for coordination between policy domains without manual intervention. We have evaluated this system using real job traces from Intrepid and Eureka, the production Blue Gene/P and data analysis systems, respectively, deployed at Argonne National Laboratory. Our experimental results quantify the costs of coscheduling and demonstrate that coscheduling can be achieved with limited impact on system performance under varying workloads.

I. INTRODUCTION

High-end computing (HEC) systems are increasingly using heterogeneous processing elements in their designs. In the typical configuration, accelerators such as GPUs are used in addition to the traditional CPUs. Meanwhile, the volume of data produced by high-end applications continues to grow, leading to an increasing demand for data analysis and visualization capabilities.

The trend has driven the deployment of *coupled systems*, where a general-purpose compute system running scientific computations or simulations is connected by a shared filesystem or over the network to a special-purpose system used for data analysis and visualization. Such systems are already common, such as Intrepid and Eureka at Argonne National Laboratory, Jaguar and Lens at Oak Ridge National Laboratory (ORNL), and Ranger and Longhorn at the Texas Advanced Computing Center (TACC). Figure 1 depicts a typical coupled HEC system. On such systems, many applications run in a post hoc fashion: data generated by compute applications is first written to storage systems and then processed by analysis applications.

While post hoc execution is common, however, co-execution is increasingly demanded. One reason is that co-execution enables monitoring of simulations, debugging, and visual debugging of the simulation code at run time. Another reason is that co-execution can accelerate the I/O time [33]—one of the critical bottlenecks faced by simulations—by staging simulation I/O data to the memory of a couple resource and avoiding writing data to persistent storage. This strategy requires that both compute and analysis applications be alive at

the same time. Furthermore, computations in several scientific domains require access to heterogeneous resources to concurrently execute models wherein one of more models are tailored for GPU-based systems while others are optimized for CPU-based clusters. This kind of computation requires co-execution spanning heterogeneous coupled resources. Thus, many existing applications already can benefit from co-execution on coupled systems, and more are anticipated if job co-execution can be conveniently achieved.

Resource managers commonly used in high-performance computing systems support a rich set of descriptions for resource requirements. However, capabilities for representing interjob temporal constraints are lacking. In general, job-ordering constraints (modeled as job dependencies) are supported, but job co-execution is not. This latter capability is a prerequisite for the execution model described above.

In this paper, we propose a coscheduling mechanism that coordinates execution between jobs on different systems. Specifically, the coscheduling can guarantee that associated jobs (e.g., a compute job with the associated data analysis job) start simultaneously across the coupled systems. The mechanism is built on top of a lightweight protocol for coordination across policy domains without manual intervention. Unlike the commonly known coscheduling mechanisms to coordinate process co-execution within a cluster of time-sharing machines [21], our coscheduling involves scheduling coordination among multiple independent resource management domains.

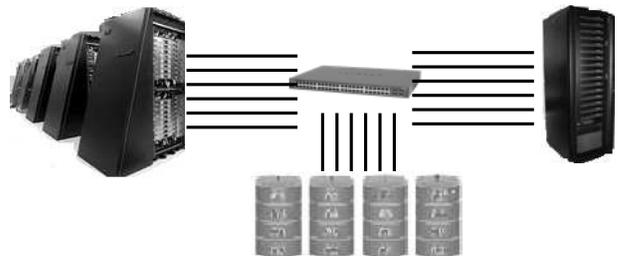


Fig. 1. Typical coupled HEC systems: a large-scale compute system (left) and a special data analysis/visualization system (right), which share a single storage system via a network.

The primary challenge of such coscheduling comes from the combination of multiple scheduling policies on multiple resource management domains. Minimizing the coupling of the multiple resource managers involved in coscheduling increases scalability and enables the implementation of this approach across multiple resource managers. Coscheduling also costs more than independent scheduling; thus, minimizing this cost is also another clear goal.

To achieve these goals, we designed a distributed algorithm and implemented it in an existing resource manager. We evaluated our mechanism with event-driven simulations using real job traces from the production Blue Gene/P [7] and data analysis systems deployed at Argonne National Laboratory. Our experimental results demonstrate that coscheduling can be achieved with limited impact on system performance under varying workloads. We also quantified the performance sensitivity of coscheduling to various configurations.

The remainder of this paper is organized as follows. Section II presents background, including some sample coupled systems and motivating applications. Section III discusses some relevant work. Section IV describes our coscheduling design. Section V evaluates coscheduling performance via trace-based simulations. Section VI concludes the paper and points out future work.

II. BACKGROUND

We begin with a discussion of several existing coupled systems. We then describe a few applications that motivate our study of coscheduling.

A. Examples of Coupled Systems

Many supercomputing centers have a large-scale system in charge of simulation and another system responsible for data analysis and visualization. One example is Intrepid and Eureka at Argonne National Laboratory.

Intrepid is a 556 TF Blue Gene/P system [7] operated by the Argonne for the U.S. Department of Energy INCITE program [2]. The system comprises 40,960 quad-core nodes, giving a total of 163,840 cores. It was ranked 13th in the latest Top500 list released in November 2010 [6]. Eureka is a data analysis cluster, comprising 100 computing nodes with 800 Xeon cores, 3.2 TB memory, and 200 NVIDIA Quadro FX 5600 GPUs. It is currently the largest installation of GPUs. A Myrinet switch complex connects Intrepid, Eureka, and a storage system consisting of PVFS [19] servers and storage nodes. Therefore, Intrepid and Eureka can exchange data on the shared file system or through the network directly.

Other such deployments exist in other supercomputing centers. For example, TACC has Ranger and Longhorn. Ranger is a SunBlade system with 62,976 cores, currently ranked 15th in the Top500 list. Longhorn is a 256-node Dell cluster with 128 GPUs. ORNL deploys Jaguar and Lens. Jaguar is a Cray XT5 system with 224,162 cores, currently ranked second in the Top500 list. NICS/UTK has Kraken and Verne. Kraken is a Cray XT5 with 98,928 cores, ranked eighth in the Top500. Verne is a 5-node Dell cluster.

B. Motivating Applications

A variety of applications would benefit from coscheduling. One is the coupled type in which computing and data analysis or visualization are conducted separately. For example, FLASH [30] is used to simulate buoyancy-driven turbulent nuclear burning; it uses VL3 [9] for visualization. PHASTA [15] is used for parallel, hierarchic, adaptive, stabilized transient analysis; it uses the visualization tool ParaView [5]. Both can benefit from co-execution. Specifically, running simultaneously with the compute application, the analysis/visualization application can process the output data at run time. Furthermore, if both applications are alive, the two can exchange a large amount of data via the network instead of using a permanent storage system, thus accelerating the I/O time [33]. Currently, co-execution is fulfilled mainly by making a reservation manually or submitting one job immediately after the related job starts (it works only when one of the systems is lightly loaded).

Another type of motivating application is the coupled computation using heterogeneous resources across the different systems. An example is the weather forecasting models run by NASA [4] wherein multiple climate analysis models are executed concurrently and their results are fed into one or many prediction models where each model is typically an independent parallel program. A key requirement to attain real-time climate prediction during hurricanes is to schedule the various executable concurrently. Moreover, some of the models could be optimized to run on GPU-based systems while others are tailored for CPU-based systems.

Furthermore, with coscheduling, some existing work in more loosely coupled systems, such as Grid [13], can also be applied to our coupled HEC system. One example is MPICH-G2 (Globus-enabled MPI) [18], which provides for MPI-style interprocess communication between computing resources. Moreover, applications similar to metacomputing [10] can also be applied to the coupled HEC environment with coscheduling.

In summary, a number of existing applications can benefit from coscheduling; and in return, coscheduling can boost the emergence of more applications taking advantage of the heterogeneous resources in coupled HEC systems.

III. RELATED WORK

The term “coscheduling” is commonly used to denote a specific mechanism proposed for concurrent systems that schedules related processes to run on different processors at the same time [21]. It has a strict version named gang scheduling [34] and other versions or enhancements proposed in the literature, such as demand-based coscheduling [25], dynamic coscheduling [24], buffered coscheduling [22], and flexible coscheduling [12].

These mechanisms share a similar goal with our coscheduling: running related processes (in our case jobs) simultaneously. The difference is that the former are used for time-sharing systems belonging to a single scheduling domain,

where nodes and job queues are managed by a single resource manager. Driven by the use of coupled systems, our coscheduling fulfills the need to start associated jobs across multiple scheduling domains (i.e., different resource managers with independent scheduling policies).

Advance resource co-reservation has been widely proposed to coordinate resource allocation across multiple systems. MacLaren [20] presented the Highly-Available Resource Co-allocator (HARC), a system for creating and managing resource reservations used for metacomputing applications [10] or workflow applications. Foster et al. [14] proposed GARA, a general-purpose architecture for reservation and allocation. Yoshimoto et al. [35] presented GUR, a system for coscheduling compute resources in a Grid computing environment using user-settable reservations similar to travel arrangements.

While co-reservation can be used to start related jobs on multiple systems at the same (reserved) time, however, it is not suitable in coupled HEC systems. First, because the two coupled systems are administratively heterogeneous, co-reservations on both machines involve expensive manual efforts in policy negotiation. Second, excessive use of reservation will leave temporal fragmentations on the computing resources, thereby leading to worse response times for regular jobs [26]. Our work fits the coupled HEC environment well because it is free from manual intervention and will leave no temporal resource fragmentation.

Metascheduling [32] also involves scheduling jobs on multiple clusters. It aims at optimizing computational workloads by combining an organization's multiple distributed resource managers into a single, aggregated view, allowing batch jobs to be directed to the best location for execution. Existing work includes GridWay by Globus Alliance [16], LoadLeveler by IBM [17], and Moab by Adaptive Computing Inc. [3]. However, these schedulers require a global job submission portal. A unique feature of our work is that we remove the restriction of a global submission portal by enabling independent job submission on each resource.

The term coscheduling is also used in other problem domains with various objects, such as coscheduling of computation and data [23], coscheduling CPU and network capacity [8][27], and coscheduling CPUs and GPUs for heterogeneous computing [29]. The object of our coscheduling is associated jobs that need co-execution on different machines in coupled HEC systems.

IV. COSCHEDULING DESIGN

In this section we define job scheduling, present our core coscheduling algorithm and its various configurations, and discuss enhancements and potential drawbacks.

A. Problem Statement

Suppose two systems (machines) A and B are running workloads of parallel jobs. Jobs on each system are managed by different resource managers R_1 and R_2 , respectively. R_1 and R_2 use independent scheduling policies. Among all the jobs on the two systems, there exist some pairs of associated

jobs. In such a job pair, one job is submitted to one system, and the other job to the other system. These two associated jobs need to be started at the same time even though they are scheduled separately on different systems.

We need a coscheduling mechanism to guarantee all the associated jobs in the same pair start at the same time without manual reservation. Meanwhile, the mechanism must limit the side effect on system utilization and the response times of both paired and nonpaired jobs.

To fulfill coscheduling, we need to make the resource managers aware of the information of associated jobs. More essentially, we need a protocol to coordinate and synchronize the scheduling of the associated job.

B. Basic Schemes

In job scheduling, we say that a job is “scheduled,” or “ready,” when a job is selected by the scheduler to start next. A scheduled job can have the highest priority or have been given the opportunity of backfilling [31]. The job should be assigned with a designated number of nodes. Normally, when a job is scheduled, it can start immediately. With coscheduling, however, a scheduled job may not be started because it may need to wait for its mate job.

When a job is ready to run but its remote mate cannot, we have two basic coscheduling schemes, “hold” and “yield,” to choose from. With hold, when a job is ready but its mate job is not, it will hold the needed computing nodes, not allowing them to be used by others, until the remote mate gets ready in the future. With yield, it will give up the chance to run without occupying any nodes and allowing the scheduler to schedule another job.

Regarding minimizing paired job synchronization, hold is better than yield. A holding job will start immediately once its mate job gets ready. However, a job that once yielded may not necessarily be able to start when its mate is ready, because of a lack of resources, so it may need to yield repeatedly.

Regarding impact to system utilization, the yielding job imposes less impact than does the holding job. When the nodes are held by a job, they cannot be used by other jobs. The scheduler treats the held nodes as busy; other jobs can neither run on them nor hold them. On the other hand, the yield scheme rarely is harmful to system utilization.

C. Main Algorithm

The core coscheduling algorithm extends the existing function in the traditional resource manager, which is invoked when a job is scheduled and ready to run. Normally, the function starts the scheduled job on the assigned nodes. But with coscheduling, additional logic will be executed before the job can actually start. Algorithm 1 describes this function, which we call *RunJob*, enhanced with coscheduling algorithm.

The algorithm is distributed: “self.xyz” calls a local function, and “remote.xyz” calls a remote function on the remote machine. Each machine runs the same algorithm with a locally configured scheme, either hold or yield. The algorithm can be enabled or disabled by a flag.

Algorithm 1: RunJob(j, N)

Input: A scheduled job j with assigned nodes N

Result: Job j either starts, or holds, or yields. Its remote mate job k , if existing, could be triggered to start under certain condition.

```
1 if cosched_enabled then
2    $k = \text{remote.getMateJobId}(j)$ 
3   if  $k$  then
4      $\text{mate\_status} = \text{remote.getMateStatus}(k)$ 
5     switch  $\text{mate\_status}$  do
6       case "holding"
7          $\text{self.startJob}(j, N)$ 
8          $\text{remote.startJob}(k)$ 
9       endsw
10      case "queuing"
11      case "unsubmitted"
12         $\text{mate\_started} = \text{remote.tryStartMate}(k)$ 
13        if  $\text{mate\_started}$  then
14           $\text{self.startJob}(j, N)$ 
15        end
16        else
17          if  $\text{self.scheme} == \text{"hold"}$  then
18             $\text{self.holdJob}(j, N)$ 
19          end
20          if  $\text{self.scheme} == \text{"yield"}$  then
21             $\text{self.yieldJob}(j)$ 
22          end
23        end
24      endsw
25      case "unknown"
26         $\text{self.startJob}(j, N)$ 
27      endsw
28    endsw
29  end
30 else
31    $\text{self.startJob}(j, N)$ 
32 end
33 end
34 else
35    $\text{self.startJob}(j, N)$ 
36 end
```

As shown in Algorithm 1, if coscheduling is not enabled or the mate job is not found, the ready job will start normally (lines 35 and 31, respectively), skipping the coscheduling logic. If coscheduling is enabled and a valid mate is found, we will first get the status of the mate, based on which the job will act. If the remote job is in hold status, both jobs will be started immediately (lines 7 and 8). If the mate job is waiting in the queue or unsubmitted, a remote function will be called to try to run the mate job. Function $\text{remote.tryStartMate}(k)$ (line 12) invokes an additional scheduling iteration on the remote machine and returns *True* only if the mate job k gets started. If the mate job is started, the local job is also

started (line 14). If the mate job cannot run at this moment (including the unsubmitted case), the local job will either hold (line 17) or yield (line 20) according to the preconfigured local coscheduling scheme. Function $\text{self.holdJob}(j, N)$ sets job j to holding status and marks the nodes N busy, not allowing another job to use them. Function $\text{self.yieldJob}(j)$ invokes an additional local scheduling iteration to try to run other jobs. If the remote status is unknown, the algorithm continues to start the local job (line 26).

The algorithm is fault-tolerant: a job will not wait forever when the remote machine or its mate job is down. If the remote system is down, line 2 will return nothing so that the ready job will start immediately. If the mate job fails alone, the mate status will be returned as unknown (line 25). The ready job thus will start normally, too.

D. Scheme Combinations

In order to apply coscheduling on the coupled systems, each machine must be preconfigured with a coscheduling scheme. To this end, we have identified four combinations of configuration: hold-hold, yield-yield, hold-yield, and yield-hold. We discuss how these combinations work to achieve coscheduling. We also point out potential problems.

1) *Hold-Hold*: Hold-hold means using the hold scheme on both machines of the coupled system. In this setting, when one job of the paired job gets ready, it will enter hold status. Once the second job gets ready, both jobs can start immediately.

Hold-hold is effective for synchronizing two jobs, but it may result in deadlock. Indeed, theoretically, hold-hold meets all of the four conditions causing deadlock: mutual exclusion (a node can be assigned only to a job), hold and wait (a job holds some nodes and waits until its mate is ready), no preemption (we do not have preemption), and circular wait (both machines use hold, so that circular wait is possible). Figure 2 shows a simple example of deadlock. Deadlock can be solved by our enhancement discussed in next subsection.

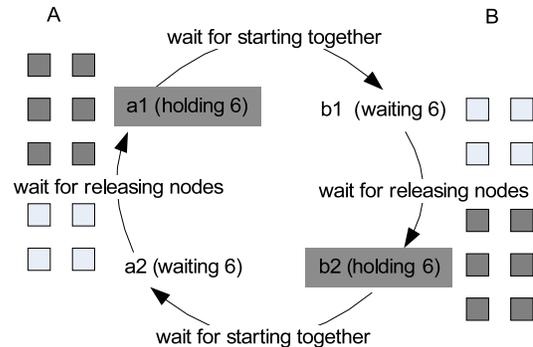


Fig. 2. Example of deadlock. Machine A has a job a_1 holding 6 nodes, waiting for its mate b_1 queuing on machine B and also requesting 6 nodes. But machine B has another job b_2 holding 6 nodes, waiting its mate a_2 queuing on machine A and requesting 6 nodes.

2) *Yield-Yield*: Yield-yield means using the yield scheme on both machines of the coupled system. In this setting, the paired jobs can be started only when both get scheduled and assigned with sufficient nodes simultaneously. Before that, both jobs may alternately yield. But they can eventually reach the both-ready condition, because they will eventually get the highest priority on their respective machine if job priority increases by time. The most commonly used FCFS (first-come, first-served) policy [11] and some of its variations such as WFP [28] serve this purpose well.

3) *Hold-Yield and Yield-Hold*: Hold-yield and yield-hold use different schemes on different machines. In this setting, the goal of coscheduling can also be achieved. Suppose machine A uses hold and machine B uses yield. If job a on A gets ready first, it will hold; when its mate job b gets ready, they can both start immediately. If job b is ready first, it will yield; when job a gets ready, it checks whether b is ready. If it is, both start; if not, job a holds until job b gets ready for the next time.

E. Enhancements

In this subsection we discuss an enhancement to solve deadlock. We also address the drawbacks of both the hold and yield schemes.

1) *Solving Deadlock*: To solve deadlock, we deploy a scheme to force the holding jobs to release their resources periodically (e.g., every 20 minutes) so that other waiting jobs can use the previously held resources. This preemptive scheme can break circular wait. Consider the same example in Figure 2. If job a_1 releases its resource temporarily, job a_2 can get the nodes to start; so can b_2 . When b_2 completes, job b_1 can start. Thus the deadlock is solved. In order to guarantee the nodes released by job a_1 can be used by job a_2 , the job that releases the nodes (i.e., a_1) should be set to the lowest priority at that scheduling iteration. If the released nodes are preempted by other jobs, the original holding job will be put in queuing status. Otherwise, the job will hold by the original holding job again.

Since deadlock can be solved, all the four configuration combinations can achieve coscheduling. That means the schemes are locally configured: an individual machine needs to be configured only with its local scheme, without knowing the remote configuration. This model makes the coscheduling schemes highly practical and scalable.

2) *Reducing Performance Impact*: To soften the impact of coscheduling on system performance, we have deployed several enhancements.

In order to reduce system utilization waste as well as regular job waiting time, it is desirable to avoid having most of the computing nodes in hold status. Therefore, we enforce a maximum threshold for the proportion of nodes. If a job is going to hold some more nodes so that proportion of held nodes exceeds the threshold, the job will yield instead of hold. With the strategy, the system can have at least a number of nodes able to be consumed normally.

To restrict the number of alternate yields, we introduce a maximum yielding threshold. That is, if a job yields more than a certain amount of times, it can start holding. Another option is to increase the priority of the job after it yields each time.

V. EVALUATIONS

In this section, we evaluate coscheduling by simulations using the job traces collected from the production coupled HEC systems deployed at Argonne.

A. Experiment Setup

To simulate coscheduling, we have extended Qsim [28], the event-driven simulator along with the Cobalt resource manager [1], to support multi-domain coscheduling simulation. For simulation, we need only to replace the real resource managers with the event-driven simulators, which maintains jobs from job traces and compute nodes from the configuration files.

In the experiment, we use real system configurations; that is, 40,960 nodes on Intrepid and 100 nodes on Eureka are available for scheduling. The job traces were collected from the production Intrepid and Eureka systems within the year of 2010. On Intrepid, the job size ranges from 512 nodes to 32,768 nodes. On Eureka, the job size ranges from 1 node to 100 nodes.

The scheduling policy used on both schedulers is the same as those used on the real machines, namely, WFP [28] plus backfilling. The coscheduling algorithm can be plugged in and out, so that we can compare the performance with and without coscheduling. To break the hold-hold deadlock, we set the held nodes' releasing period at 20 minutes.

B. Capability Validation

We ran a large number of simulation cases to verify the coscheduling mechanism. A simulation case is determined by three configurations: a combination of the scheme configurations, a combination of system utilization rates on both machines, and the proportion of paired jobs. We tuned the scheme configuration for four options: hold-hold, hold-yield, yield-hold, and yield-yield. We also tuned the system utilization rate and proportion of paired jobs to various values. We ran each case 10 times.

All simulation cases ran successfully to completion, and the output logs show that all the paired jobs start at the same time with their own mate jobs no matter which one gets ready first. That means that coscheduling is achieved by all configurations of schemes under various workloads.

Furthermore, deadlock is solved with the enhancement described in subsection IV-E1. Without the enhancement, deadlocks are highly likely to be observed when the simulation time span more than 10 days. When deadlock occurs, the job queues on both machines keep growing, but no job can start. With the enhancement, this phenomenon never happens. In the experiment, we set the releasing interval to 20 minutes, which can be tuned freely by system owners.

The other enhancements turned out to be optional to achieve coscheduling. The simulations ran successfully when we set

the nodes that could be used for holding as the whole system. Even when the threshold was not for maximum yielding times, no starvation was observed. That is, all the paired jobs on the system using the yield scheme start naturally in the scheduling simulation, without waiting until in the cooling-down phase of the simulation.

Although coscheduling can be achieved for all simulation cases, the performance varies with configuration. In the rest of this section, we briefly describe the metrics used and then present a set of results to quantify the performance and cost of the coscheduling under some typical system loads and proportion of paired jobs.

C. Evaluation Metrics

Four metrics were used in the performance evaluation.

- *Waiting time (wait)*: the time period between when the job is submitted and when it is started.
- *Slowdown (slowdown)*: a job's response time (waiting time plus running time) divided by the job's running time. It captures the fact that a long job can endure longer waiting than a short job can. The average *wait* and *slowdown* among all the jobs can measure the overall system performance of job scheduling.
- *Paired job synchronization time (sync time)*: the extra time a job has to wait for its mate in a coscheduling setting. By examining the average synchronization time among all the paired jobs, we can measure the performance cost to the paired jobs.
- *Service unit loss*: the wasted computing capabilities caused by holding. We measure this metric in node-hours and system utilization rate. This metric reflects the impact of coscheduling on system utilization.

D. Performance under Various System Loads

In this part of our experiments we measure the performance impact brought by coscheduling under different Eureka system load. Because the current Intrepid system load is high and stable but the Eureka system load low and unstable (it is highly likely to vary in the future), we wish to study how coscheduling behaves when the Eureka load varies. This information will help us decide the way to choose job traces.

We set the time span as one month. For Intrepid, we used the real job trace in a month containing 9,219 jobs. For Eureka, we used half-synthetic job traces to simulate various workload on Eureka. That is, by adjusting the job arrival intervals we can pack the workload in multiple months into one month.

We made three new Eureka traces with system utilization rates, 0.25, 0.5, and 0.75, to present low, medium, and high system load, respectively. For making each new trace, we multiplied a same fraction to each job arrival interval in the real Eureka trace, so that the shape of job arrival distribution was the same with the real trace. In each simulation, we associated the two jobs on different machines if their submission times were within 2 minutes. The resulting proportion of paired jobs was between 5% and 10%.

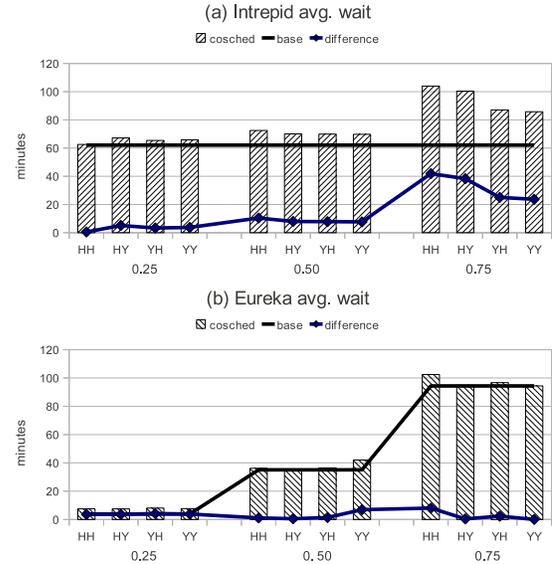


Fig. 3. Scheduling performance (avg. wait) by Eureka system load.

Now we will present the results. Figure 3 shows the average waiting time. The bars on the x-axis represent simulations using coscheduling. They are in three groups. Each group represents a system load on Eureka, measured by a system utilization rate of 0.25, 0.50, and 0.75, respectively. Within each group are four bars, representing the coscheduling scheme combinations: HH means using hold on both machine, YY means using yield on both machines, HY means using hold on Intrepid and yield on Eureka, and YH means using yield on Intrepid and hold on Eureka. The y-axis shows the average waiting minutes of total jobs.

Figure 3(a) shows the average waiting times for Intrepid. The horizontal line represents the baseline (61 minutes) that is simulated without coscheduling (or no paired jobs at all). The bar values are all above the base, meaning that using coscheduling degrades the overall performance with respect to average waiting time. The line with points on it shows the difference between coscheduling results and the base. Specifically, under low (0.25) and medium (0.50) system loads, the coscheduling causes extra average waiting of less than 4 minutes and 10 minutes, respectively. But under high load (0.75), the difference grows to 42 minutes.

Figure 3(b) shows the average waiting times for Eureka. In this figure, each bar group has its own baseline because the system utilization rate on Eureka varies. No matter what the system utilization rate is, the differences between coscheduling results and bases are small. YY under 0.50 has 7 minutes more average waiting, and HH under 0.75 has 8 minutes more; other results with coscheduling are all under 5 minutes.

Figure 4 shows the average slowdown for both Intrepid and Eureka. The trend of these results is similar to that for the average waiting time. The base slowdown on Intrepid is 6.1, which means an average job's response time on Intrepid is 6.1 times its running time. The base average slowdowns on

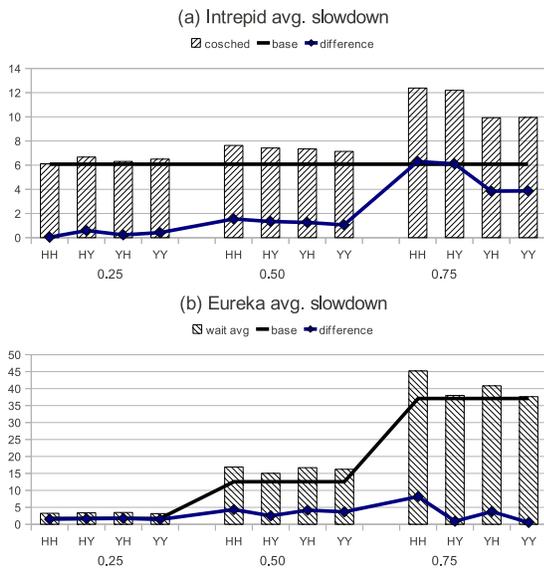


Fig. 4. Scheduling performance (avg. slowdown) by Eureka load.

Eureka are 1.7, 12.5, and 37.1 for three workloads.

For Intrepid, the increase average slowdown is under 0.6, 1.5, and 6.3 for the three groups. Only under the high Eureka load does the slowdown on Intrepid show a significant increase; but the absolute value are all under 12.4, which is not high. For Eureka, the slowdowns of coscheduling are fairly close to those of the bases.

Combining Figure 3 and Figure 4, we can make the following observations. First, the impact of coscheduling on overall system performance is reasonably low. On Intrepid, the increased average waiting time is mostly under 10 minutes, with only one exception under high Eureka load. Even for that case, if we avoid using hold on Intrepid, we can limit the increased average waiting to under 25 minutes. Eureka requires even less extra waiting time: most times are under 5 minutes, and the maximum time is 8 minutes. Slowdown has a similar trend. Therefore, the results indicate that we can deploy the coscheduling mechanism on current systems with low impact.

Second, for medium and high system loads, we observe that using hold has worse waiting time or slowdown than using yield under the same system utilization and same remote scheme. This result suggests that although using hold can cause less synchronization time for those paired jobs, it impacts resource utilization by holding some idle nodes, so that other regular jobs will suffer more waiting time; thus, the overall averages of wait and slowdown are affected. Under low system load, however, this trend is not observed. The main reason is that consuming resources does not have much effect on waiting jobs when the system load is low.

Figure 5 shows the average paired job synchronization times. There are six groups of bars. Each group is labeled on the x-axis by a configuration combining the system utilization rate (0.25, 0.50, and 0.75) with the remote machine scheme

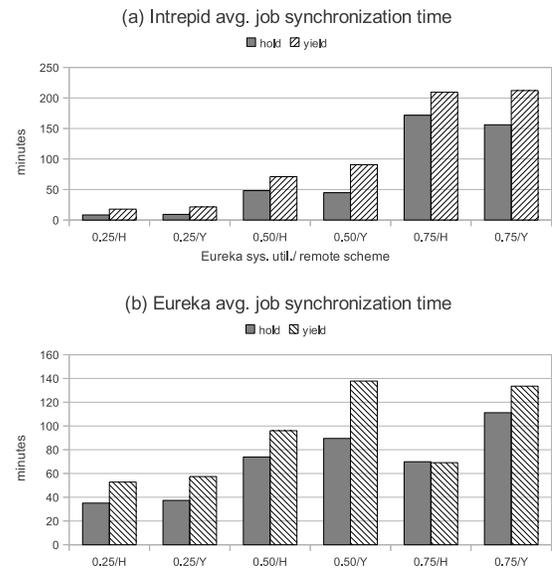


Fig. 5. Average paired job synchronization time by Eureka load.

(H or Y). Within each group are two bars, each representing a coscheduling scheme on the local machine (H or Y), indicated by the legend.

Figure 5(a) shows the results for Intrepid. When Intrepid uses hold, the average job synchronization time varies under different Eureka system utilization. As the system load gets higher, the sync time increases, from under 10 minutes to nearly 50 minutes, then to above 150 minutes. Comparing bars within each group, we see that using yield on the local machine results in more average sync time than using does hold, given that the remote scheme and system load are the same. This result is consistent with the design of these two schemes: hold is better regarding extra waiting time than yield is.

Figure 5(b) shows the results for Eureka. When Eureka uses hold, the average job sync time varies from 35 to 111 minutes. When Eureka uses yield, the overhead varies from 52 to 137 minutes. In the same group, using hold consumes less overhead than using yield, except for the 0.75/H group, in which the two bars are almost the same.

Figure 6 shows the service unit loss on both Intrepid and Eureka. The x-axis is the same as in the previous figure. The primary y-axis shows the service unit loss measured by node-hours. The secondary y-axis shows the corresponding system utilization rate loss compared with the total system node-hours over the whole time span. The figures show only the loss caused by using hold on the local machine.

Figure 6(a) shows the results for Intrepid. When system load gets higher, the lost node-hours on Intrepid in a month increase from 135,000 to 1.2 million node-hours. In terms of system utilization rate, they represent 0.46% to 4.6% of the total node-hours in a month on Intrepid.

Figure 6(b) shows the results for Eureka. The lost node-hours appear unrelated to the system utilization rate. The high load case causes the least node-hour loss. Specifically, the

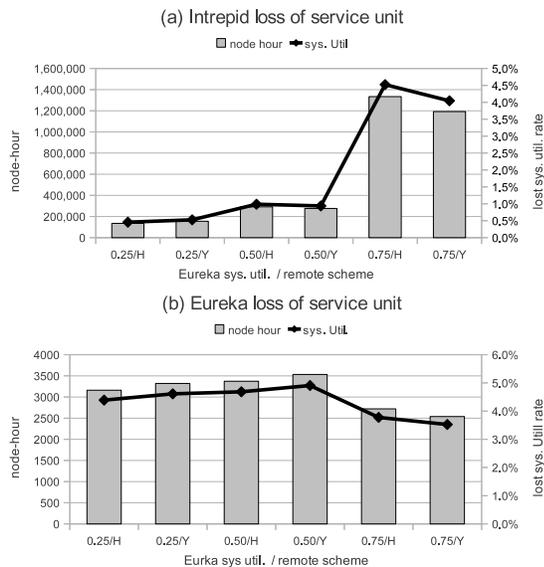


Fig. 6. Service unit loss by Eureka load.

total node-hour loss in a month ranges from 25,000 to 35,000, which corresponds to 3.5% to 4.9% of the total node-hours in a month on Eureka.

As indicated in Figure 5 and Figure 6, using the hold scheme results in less average synchronization time for paired jobs than using yield, but it causes extra loss in system utilization. This trade-off needs to be balanced by system owners based on system priorities and user needs.

E. Performance under Various Paired Job Proportions

We conducted several simulations to explore the impact of coscheduling under different proportions of paired jobs. This study can provide insight to system owners to control the coscheduling configuration under certain conditions.

For Intrepid, we use the same job trace as used previously. For Eureka, we generate a special workload that has the same number of jobs and is within the same time span as the Intrepid trace. By doing so, we can conveniently tune the proportion of paired jobs on both traces. The system utilization rate of this special Eureka workload is around 0.5, representing the medium load. For the various simulations, we set the paired job proportions to 2.5%, 5%, 10%, 20%, and 33%.

Figure 7 shows the average waiting minutes for both Intrepid and Eureka under different proportions of paired jobs. The x-axis is similar to that of Figure 3; the only difference is that the system utilization rates in Figure 3 is replaced here by proportions of paired jobs.

Figure 7(a) shows the average waiting times for Intrepid. Clearly, the higher the paired job ratio is, the more the relative waiting time increase is on Intrepid. When the paired job ratio is 10% or less, the average waiting time is under 76 minutes, meaning 3–14 minutes more than the base; the relative increase is 5%–22%. When the paired job ratio is 20%, the average waiting times increase to 90–100 minutes, or 26–38 minutes

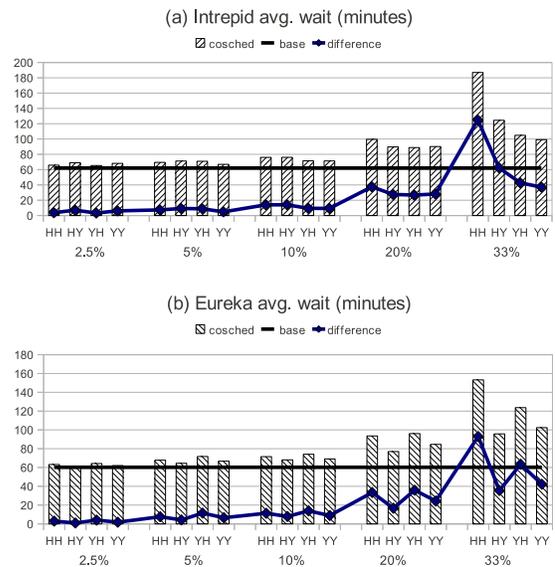


Fig. 7. Average waiting times by paired job proportion.

more than the base. When one-third of the total jobs (33%) are paired, the average waiting time gets considerably worse with the hold scheme (up to 187 minutes). If the yield scheme (YH or YY) is used, the average waiting in this case is comparable to the case with 20% paired jobs (around 100 minutes).

Figure 7(b) shows the average waiting times for Eureka. Similarly, the higher the paired job ratio is, the more the relative waiting time increase is on Eureka. The trend is similar to that of Intrepid. The overall performance is not significantly impacted when the proportion of paired jobs is under 20%, regardless of which coscheduling scheme is used. With 33% paired job proportion, using hold (HH and YH) can cause a noticeable performance degradation, and using yield (HY and YY) can achieve performance similar to the 20% proportion case.

Figure 8 shows the average slowdown for both Intrepid and Eureka under different proportions of paired jobs. Figure 8(a) shows the results for Intrepid. The trend is similar to the waiting times. For the first three proportions, the average slowdowns are under 7.5. For the last two relatively high proportions, the slowdowns are mostly between 7.9 to 12.9, except with hold-hold.

Figure 8(b) shows the average slowdown for Eureka. Again, the higher the paired job ratio is, the more the relative waiting time increase is on Eureka. For the first three proportion cases, the difference in average slowdown is only in the single digits. For the last two relatively high proportion cases, the average slowdown shows a double-digit increase. Again, hold-hold is the worst case under the high system load.

Comparing Figure 7 and Figure 8, we can see that coscheduling will not impact system performance significantly with 20% proportion of paired jobs. When the paired jobs proportion gets higher than one-third (33%), however, the performance degradation is noticeable. We recommend disabling

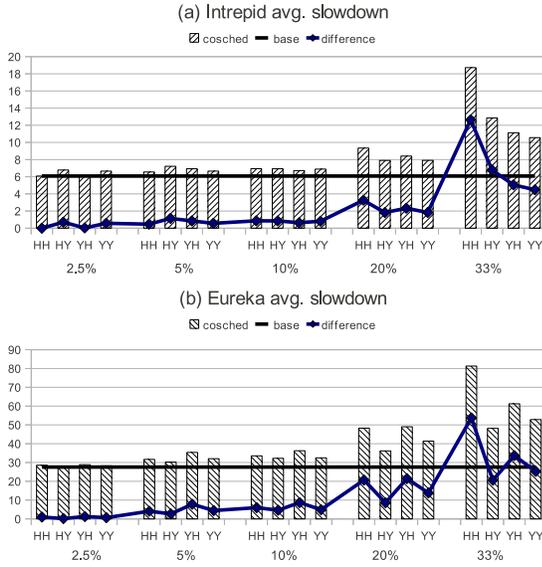


Fig. 8. Avg. slowdowns by paired job proportion.

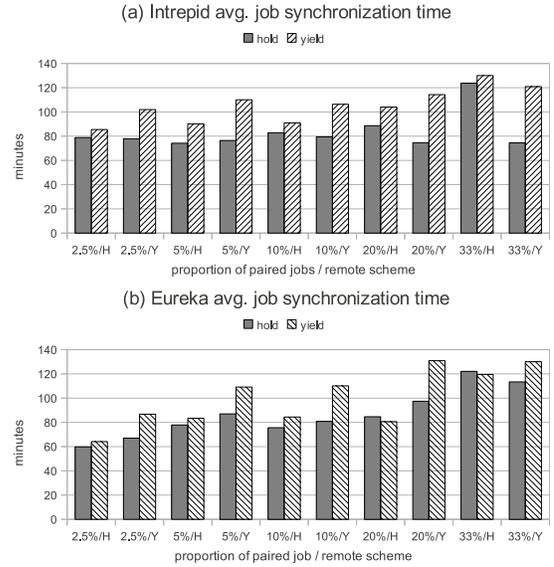


Fig. 9. Paired job average synchronization time by paired job proportion.

the coscheduling feature under this condition, or at least not using the hold scheme. Indeed, as shown in the figure, when the proportion is 10% or more, the hold scheme shows more negative impact than does the yield scheme.

Figure 9 shows the average paired job synchronization time for both Intrepid and Eureka. The x-axis is similar that in Figure 5 except that the system utilization rate is replaced by the proportion of paired jobs.

Figure 9(a) shows the results for Intrepid. When Intrepid uses hold, the average sync time mostly ranges between 75 and 88 minutes, except for HH with 124 minutes. The range is narrower than that in Figure 5(a), indicating that the average sync time for paired jobs is more sensitive to the system load than to the proportion of job pairs. But here, too, the observation applies that using hold as the local scheme consumes less average sync time than using yield.

Figure 9(b) shows the results for Eureka. When Eureka uses hold, the average paired job sync time varies from 60 to 122 minutes. There is a slight trend for overhead to increase as the paired job proportion increases, but the slope is not as significant as in Figure 5(b). Similarly, all the cases using hold as the local scheme are better than using yield in terms of sync time.

Figure 10 shows service unit loss on both Intrepid and Eureka. On both machines, the loss of service unit increases as the proportion of paired jobs gets higher. Specifically, the results on Intrepid range from 200,000 to 2.7 million node-hours, or 0.7% to 9.3% of the total node-hours in a month; the losses on Eureka range from 800 to 15,000 node-hours, or 1% to 21% of total node-hours in a month.

We consider the service unit loss acceptable when the proportion of paired jobs is under 10%, when the loss rate is under 3% on Intrepid and 5% on Eureka. When the proportion is 20%, the loss rate is also fairly acceptable. But when paired

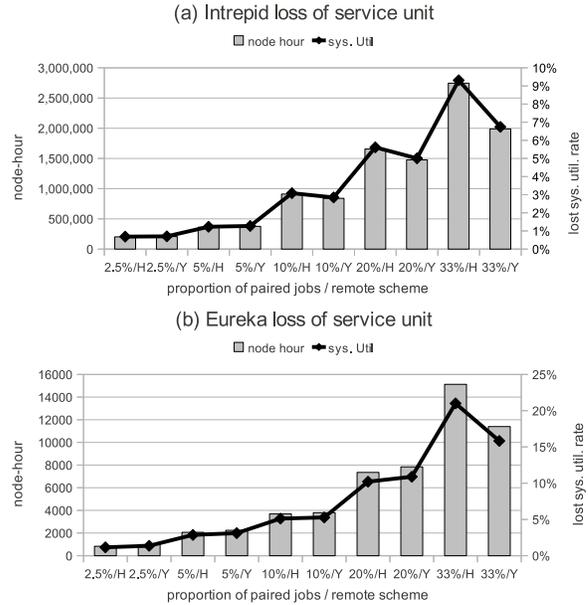


Fig. 10. Service unit loss by paired job proportion.

job become more than 33%, if the coscheduling feature cannot be disabled, we recommend using the yield scheme on both machines in order to avoid loss of service units.

VI. SUMMARY

We have designed and implemented a coscheduling mechanism used in a coupled high-performance computing system where a large-scale computing system co-resides with a special-purpose data analysis or visualization system. The goal of coscheduling is to simultaneously start associated jobs that are submitted to different machines belonging to

different scheduling domains. We have proposed two basic schemes for coscheduling: hold and yield. Any combination of schemes on the coupled systems can achieve the coscheduling goal without manual intervention. Our algorithm is practical and scalable because of the lightweight protocol and simple interfaces. It facilitates systems using different resource managers or schedulers to interface in order to coschedule jobs. For example, with our protocol, jobs submitted to a compute resource running LSF can be coscheduled with job submitted to an analysis resource running PBS.

We have evaluated the coscheduling mechanism by means of trace-based simulations using real configurations and workloads from the production coupled systems deployed at Argonne National Laboratory. Our experimental results demonstrate that coscheduling can be achieved with limited impact on system performance under varying workloads. The quantified results of coscheduling performance under specific configurations and system conditions are instructive to system owners in using coscheduling.

Achieving coscheduling not only benefits a number of existing applications using the coupled systems but also can drive the emergence of new applications to take advantage of (administratively) heterogeneous resources. In the future, we plan to extend our coscheduling mechanism to support more sophisticated inter-job temporal constraints. Further, we will examine the possibility of extending our algorithm to support N-way coscheduling on more than two scheduling domains. Ultimately, we plan to deploy our coscheduling mechanism on the production systems at Argonne and measure the benefit to real applications.

ACKNOWLEDGMENTS

The work at IIT is supported in part by U.S. NSF grants CNS-0834514, CNS-0720549, and CCF-0702737. We gratefully acknowledge the use of the resources of the Argonne Leadership Computing Facility at Argonne National Laboratory. The work at Argonne National Laboratory was supported by the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract DE-AC02-06CH11357 and an Argonne National Laboratory Director's Postdoctoral Fellowship.

REFERENCES

- [1] Cobalt resource manager. <http://trac.mcs.anl.gov/projects/cobalt>
- [2] DOE INCITE program. <http://www.er.doe.gov/ascr/incite>
- [3] Moab resource manager. <http://www.adaptivecomputing.com>
- [4] NASA MAP program. <http://map.nasa.gov/>
- [5] ParaView—Scientific Visualization. <http://www.paraview.org>
- [6] TOP500 Supercomputing web site, <http://www.top500.org>
- [7] Blue Gene Team, "Overview of the IBM Blue Gene/P project," *IBM Journal of Research and Development*, 2008.
- [8] J. Basney and M. Livny, "Improving goodput by co-scheduling CPU and network capacity," *International Journal of High Performance Computing Applications*, 13(3), 220–230, 1999.
- [9] J. Binns, F. Dech, M. Papka, J. Silverstein, and R. Stevens, "Developing a distributed collaborative radiological visualization application," in *From Grid to HealthGrid*, pp. 70–79, 2005.
- [10] K. Czajkowski, I. Foster, N. Karonis, C. Kesselman, S. Martin, W. Smith, and S. Tuecke, "A resource management architecture for metacomputing systems," in *Proc. of Job Scheduling Strategies for Parallel Processing (JSSPP)*, 1998.

- [11] Y. Etsion and D. Tsafir, "A short survey of commercial cluster batch schedulers," Technical Report 2005-13, the Hebrew University of Jerusalem, 2005.
- [12] E. Frachtenberg, D. Feitelson, F. Petrini, and J. Fernandez, "Flexible coscheduling—mitigating load imbalance and improving utilization of heterogeneous resources," in *Proc. of IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, 2003.
- [13] I. Foster and C. Kesselman, *The Grid: Blueprint for a New Computing Infrastructure*, Morgan Kaufmann Publishers, 1999.
- [14] I. Foster, C. Kesselman, C. Lee, R. Lindell, K. Nahrstedt, and A. Roy, "A distributed resource management architecture that supports advance reservations and co-allocation," in *Proc. of International Workshop on Quality of Service*, 1999.
- [15] J. Fu, N. Liu, O. Sahni, K. Jansen, M. Shephard, and C. Carothers, "Scalable parallel I/O alternatives for massively parallel partitioned solver systems," in *Proc. of Int'l Parallel & Distributed Processing Symp. Workshops (IPDPSW)*, 2010.
- [16] E. Huedo, R. Montero, I. Llorente, "A framework for adaptive execution in grids," *Software—Practice & Experience* 34(7), 631–651, 2004.
- [17] IBM, "Workload management with LoadLeveler," <http://www.redbooks.ibm.com/redbooks/pdfs/sg246038.pdf>
- [18] N. Karonis, B. Toonen, and I. Foster, "MPICH-G2: A Grid-enabled implementation of the Message Passing Interface," *Journal of Parallel and Distributed Computing*, 63(5), 551–563, 2003.
- [19] S. Lang, P. Carns, R. Latham, R. Ross, K. Harms, and W. Allcock, "I/O performance challenges at leadership scale," in *Proc. of Supercomputing Conference (SC'09)*, 2009.
- [20] J. MacLaren, "HARC: The highly-available resource co-allocator," in *Proc. of GADA'07, LNCS 48(04)*, Springer-Verlag, 1385–1402, 2007.
- [21] J. Ousterhout, "Scheduling techniques for concurrent systems," in *Proc. of ICDCS'82*, 1982.
- [22] F. Petrini and W.-C. Feng, "Buffered coscheduling: A new methodology for multitasking parallel jobs on distributed systems," in *Proc. of IEEE IPDPS'00*, 2000.
- [23] A. Romosan, D. Rotem, A. Shoshani, and D. Wright, "Co-scheduling of computation and data on computer clusters," in *Proc. of International Conference on Scientific and Statistical Database Management*, 2005.
- [24] P. Sobalvarro, S. Pakin, W. Weihl, and A. Chien, "Dynamic coscheduling on workstation clusters," in *Proc. of Job Scheduling Strategies for Parallel Processing (JSSPP)*, 1998.
- [25] P. Sobalvarro and W. Weihl, "Demand-based coscheduling of parallel jobs on multiprogrammed multiprocessors," in *Proc. of Job Scheduling Strategies for Parallel Processing (JSSPP)*, 1995.
- [26] W. Smith, I. Foster, and W. Taylor, "Scheduling with advanced reservations," in *Proc. of IEEE IPDPS'00*, 2000.
- [27] N. Taesombut and A. Chien, "Evaluating network information models on resource efficiency and application performance in lambda-Grids," in *Proc. of ACM/IEEE Supercomputing Conference (SC'07)*, 2007.
- [28] W. Tang, Z. Lan, N. Desai, and D. Buettner, "Fault-aware, utility-based job scheduling on Blue Gene/P systems," in *Proc. of IEEE Int'l Conf. on Cluster Computing (Cluster'09)*, 2009.
- [29] G. Teodoro, R. Sachetto, O. Sertel, M. Gurcan, W. Meira, U. Catalyurek, and R. Ferreira, "Coordinating the use of GPU and CPU for improving performance of compute intensive applications," in *Proc. of IEEE Int'l Conf. on Cluster Computing (Cluster'09)*, 2009.
- [30] D. Townsley, R. Bair, A. Dubey, R. Fisher, N. Hearn, D. Lamb, and K. Riley, "Large-scale simulations of buoyancy-driven turbulent nuclear burning," *Journal of Physics: Conference Series*, 125(1), 2009.
- [31] D. Tsafir, Y. Etsion, and D. Feitelson, "Backfilling using system-generated predictions rather than user runtime estimates," *IEEE Transactions on Parallel and Distributed Systems* 18(6), 789–803, 2007.
- [32] S. Vadiyar and J. Dongarra, "A metascheduler for the grid," in *Proc. of 11th IEEE International Symposium on High Performance Distributed Computing (HPDC)*, 2002.
- [33] V. Vishwanath, M. Hereld, K. Iskra, D. Kimpe, V. Morozov, M. Papka, R. Ross, and K. Yoshii, "Accelerating I/O forwarding in IBM Blue Gene/P systems," in *Proc. of ACM/IEEE Supercomputing Conference (SC'10)*, 2010.
- [34] Y. Wiseman and D. Feitelson, "Paired gang scheduling," in *IEEE Trans. Parallel and Distributed Systems*, 14(6), 581–592, 2003.
- [35] K. Yoshimoto, P. A. Kavatch, and P. Andrews, "Co-scheduling with user settable reservations," in *Proc. of Job Scheduling Strategies for Parallel Processing (JSSPP)*, 2005.

The submitted manuscript has been created by UChicago Argonne, LLC, Operator of Argonne National Laboratory (“Argonne”). Argonne, a U.S. Department of Energy Office of Science laboratory, is operated under Contract No. DE-AC02-06CH11357. The U.S. Government retains for itself, and others acting on its behalf, a paid-up nonexclusive, irrevocable worldwide license in said article to reproduce, prepare derivative works, distribute copies to the public, and perform publicly and display publicly, by or on behalf of the Government.