# Time-Sharing Redux for Large-scale Systems

Steven Hofmeyr, Lawrence Berkeley National Laboratory (LBNL), shofmeyr@lbl.gov
Costin Iancu, LBNL
Juan A. Colmenares, Samsung Research America
Eric Roman, LBNL
Brian Austin, LBNL

***Abstract*—HPC facilities typically use batch scheduling to space-share jobs. In this paper we revisit time-sharing using a trace of 2.4 million jobs obtained during 20 months of operation of a modern petascale supercomputer. Our simulations show that batch scheduling produces distributions that are skewed towards smaller jobs and longer execution times, whereas time-sharing produces much more uniform slowdowns. Consequently, for applications that strong scale, the turnaround time does not scale with batch scheduling, but it does with time-sharing, resulting in turnarounds that are orders of magnitude better at the largest scales. In addition, we show that time-sharing can confer additional benefits with modern programming practices and in noisy systems. FutureExascale HPC systems, are expected to exhibit billion-way heterogeneous parallelism and poor performance predictability. As many applications will run in strong scaling, how resource allocation policies affect the experience of supercomputer users has once again become a timely subject.**

## I. Introduction

Batch scheduling (space-sharing) is currently used in all major supercomputers because it offers performance predictability for Single Program Multiple Data (SPMD) applications. However, future HPC systems are going to be much more noisy, with greatly increased concurrency and heterogeneity, which will require the use of new, asynchronous programming models. The batch scheduled environment may not be the most efficient nor the most user-friendly in this future world, so it is timely to once again explore time-sharing for HPC systems.

Previous studies [1]–[5] have shown that combining space-sharing and time-sharing for HPC workloads can be beneficial for job turnaround time (time from submission of the job to its completion). This research has focused on gang-scheduling [6], because that is necessary to efficiently run time-multiplexed SPMD codes that use fine-grain, busy-wait synchronization [7], and most scientific applications are SPMD. However, the complexity and overhead of implementing gang scheduling means that it is not typically deployed in production at large-scale HPC facilities; neither is any other form of time-sharing. By contrast, in data center and cloud computing environments, space-sharing is always combined with time-sharing, because jobs are usually non-SPMD and can be efficiently scheduled using best effort [8]–[10].

In this paper we revisit the impact of extending batch scheduling with time-sharing, for a modern petascale supercomputer, using a trace of 2.5 million jobs obtained during 20 months of operation of *Edison*, a 5576-node, 133,824-core Cray XC30 at the National Energy Research Supercomputing Center (NERSC) [11]. We simulate both first-come-first-serve (FCFS) batch scheduling with backfill (similar to Edison's scheduler) and time-sharing. For time-sharing, we assume best-effort parallel scheduling for co-located jobs [4], [5], [12], instead of gang scheduling. Our simulation takes into account resource constraints such as memory.

We provide additional insight about time-sharing over previous work by considering the implications of the distributions of job slowdowns (the normalized turnaround time [13]) under the different schedulers. We show that, on a heavily loaded system, batch scheduling produces distributions that are skewed towards larger slowdowns for bigger jobs and shorter execution times, which is precisely what happens to applications as they strong scale to more nodes. Consequently, the turnaround time for those applications does not scale as more nodes are used; rather, it gets worse at extreme scale. By contrast, time-sharing produces more or less uniform slowdowns, which means that turnaround scales as an application scales, resulting in orders of magnitude faster turnaround than batch scheduling at extreme scale.

We also investigate other important aspects of system and application configuration. System noise is likely to be a large problem as supercomputers increase in scale. Using a simple straw-man model of noise, we demonstrate that small levels of noise have a much larger impact on the slowdowns of batch scheduled jobs than time-sharing jobs. In addition, we explore the implications of modern programming practices by modeling the performance of applications that are not SPMD. We show that with time-sharing, they have about half the average slowdown of SPMD applications, and are unaffected by small levels of noise.

## II. Data

We collected data on 2,473,455 jobs submitted to Edison over a 20 month period, from January 2014 up to the end of August 2015. For the analysis we used 97% of the jobs in the trace, dropping jobs with a walltime of less than 10s, and those jobs used by system administrators for testing and benchmarking. Although 90% of jobs were under 32 nodes, half of all system utilization is taken up by jobs greater than 100 nodes, and 20% by jobs greater than 1000 nodes, i.e. one fifth of the system (Figure 1). Edison is hence a blend of capacity and capability computing, similar to many other supercomputing centers, such as TACC [14] and NCSA [15].

On average, over the 20 months studied, about 4000 jobs are submitted to Edison every day. Consequently, the utilization of the system is very high, with an average of 89% over the last 500 days of the trace. We treat the first 100 days as a warm-up

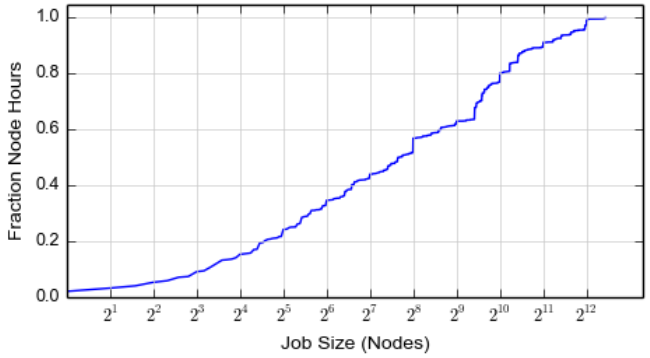


Fig. 1. Cumulative distribution function (CDF) of node hours. Half of system utilization is taken up by jobs greater than 100 nodes, and 20% by jobs greater than 1000 nodes.

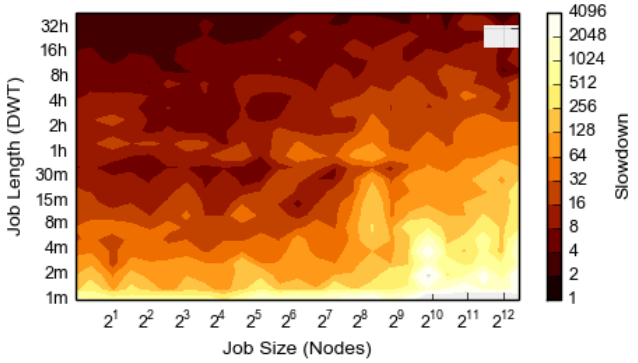


Fig. 2. Average job slowdown on Edison, binned by size and length of job. Jobs with more nodes and shorter DWT have higher slowdown.

period, to avoid the common warm-up pitfall of job scheduler simulations [16]. Saturation [17] occurs around 90%, which means that variations in utilization are largely driven by job arrival times, rather than scheduling effects [18].

We could only obtain accurate memory usage data for the 88% of jobs that use small pages. For the jobs using large-pages, we assumed that the memory usage follows the same distribution as the rest of the jobs, and used inverse transformation sampling to fill in the missing memory. We further assumed memory is independent of job size, which we validated with a Kendall's tau test [19] (coefficient of 0.0035). In general, memory usage is low, with 90% of node-hours using less than 60% of the memory per node.

To assess the impact of job scheduling on the user experience, we use *bounded slowdown* (BS) [13], which is defined as the normalized turnaround time, i.e. $BS = T/max(DWT, \beta)$, where $T$ is the turnaround time, $DWT$ is the walltime of the job when run on dedicated resources, and $\beta$ is a lower bound on the DWT. We chose $\beta = 10s$, a value commonly used in the literature to prevent very short duration jobs biasing slowdown averages. Hereafter, we use "slowdown" as shorthand for "bounded slowdown."

Using the actual submission, completion and walltimes for each job, we can compute the slowdown per job. To study the distribution, we divide the job size and the job length each into 26 logarithmic bins, and calculate the average slowdown for all jobs in a bin (Figure 2). The slowdown for a job is affected by its size and length: jobs with more nodes and shorter DWT have higher average slowdown.

## III. Batch Scheduling Simulation

We developed a discrete time simulation of a job scheduler, where each timestep is 10s. At each simulation step, we inject new jobs according to the submission time of jobs in the Edison data trace, rounded to the nearest 10s. The size and memory requirements of the observed job are used, and the length (DWT) is determined by the actual walltime of the job.

The simulated batch scheduler is first-come-first-served (FCFS) with backfill, similar to that used by Edison in production. Backfilling attempts to allocate unused nodes to smaller jobs further back in the queue whenever larger jobs at the queue's head are blocked waiting for available nodes. If the smaller job's requested walltime is less than the time the job at the head of the queue must wait, then the smaller job can run on unused nodes.

We make several simplifying assumptions for the simulation. First, we do not simulate the variety of Edison queues; we assume a single queue with no priorities. Second, we do not simulate the node reservations (e.g. on Edison, 512 nodes are reserved for the debug queue during the day, and 64 nodes during the night). Third, we assume no downtimes or periods of degraded performance, unlike the real system (there were eight scheduled outages in 2015). Finally, we assume no dedicated runs, where the system is reserved for a single user and no other jobs can run (there were six in 2015).

The simulation gives similar utilization over time to that of the Edison scheduler (Figure 3). However, the overall distribution of slowdowns is slightly better in the simulation (Figure 4), with more jobs having a slowdown of under 10 and the tail is shorter by an order of magnitude. The simulated results are not an exact match with the real data because of the aforementioned simplifications. The only cases in which the simulation does worse than the real system is for jobs in the debug queue (roughly 2-4x worse), because those jobs have access to a dedicated set of nodes.

All in all, we believe our batch scheduling simulation is sufficiently representative of the Edison batch scheduler's behavior to serve as a baseline for comparison with other scheduling policies, even with the simplifying assumptions.

## IV. Time-sharing Simulation

Our simulated time-sharing system tries to immediately start each new job on the nodes with the lowest CPU load (the number of jobs running on the node). However, some jobs may not be able to start immediately because of memory constraints. In a system such as Edison, memory does not get swapped to disk, and so it cannot be oversubscribed. Consequently, jobs that require more memory than is available will abort. To avoid this problem, we use admission control, queueing jobs that do not have sufficient memory to run.

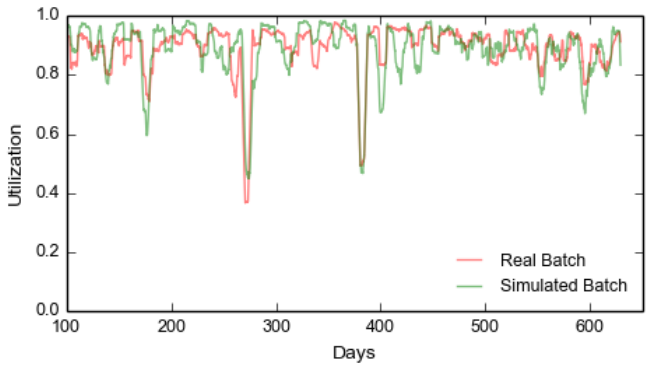


Fig. 3. System utilization over time for simulated and real batch schedulers. Shown is the last 500 days (skipping the first 100 days), and each point is a one week moving average. The simulated and real utilizations are very close, with both averaging 89% overall.

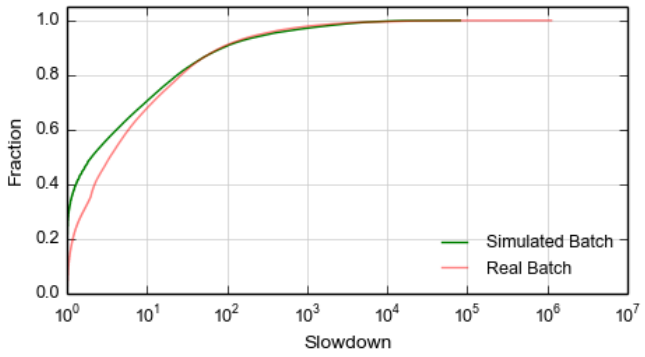


Fig. 4. CDF of slowdown for the simulated and real Edison batch schedulers. The simulated slowdown is slightly better, with more jobs of slowdown under 10, and a shorter tail to the distribution.

For admission control, we assume that the user specifies the memory requirements when submitting the job, and use the post-hoc data on memory usage from the job trace. Submitting memory estimates is no more onerous than submitting walltime estimates for batch scheduling (which are not needed for time-sharing). And like walltime estimates, a user has an incentive to submit accurate values: overestimating will delay a job, whereas underestimating will result in failure. An alternative is to estimate the memory required through inspection of binaries or historical usage [20]. We discuss other approaches to the memory pressure problem in Section VIII.

Our simulation assumes *rate equivalent* scheduling [13], where each job on a node gets an equal fraction of every time period, but there is no synchronization between the time slices across nodes. Several studies have shown that rate equivalent scheduling, such as implicit co-scheduling [12] or flexible co-scheduling [21], can achieve similar performance to gang scheduling without the concomitant problem of fragmentation. Furthermore, gang scheduling is only beneficial to fine-grain SPMD applications that synchronize using busy-wait [7], and confers no benefits when using blocking [22].

SPMD is still the most common programming model used by scientific applications, so we assume that all jobs are SPMD. Consequently, in the simulation a job runs at the speed of its slowest node, e.g., if a job is running on two nodes, one with a CPU load of two and one with a CPU load of four, then it will utilize only a $1/4$ of each node's time. The idle cycles are used by other jobs when possible, so, in the previous example, if the other job on the first node was running only on that node, it would utilize $3/4$ of the CPU cycles. In Section VII, we explore the consequences of relaxing the SPMD assumption.

When applications are time-multiplexed, sometimes they may speedup, sometimes they slow down. Sharing can improve throughput compared to running applications sequentially [23], and can provide benefits by overlapping computation and communication [13]. Slowdown comes from many sources, including the cost of context switches, the impact of cache pollution, the overhead of sharing network resources, etc. Modeling the impact of sharing, if at all feasible, requires very detailed runtime data which the traces do not contain. Consequently, in our study we run the simulations using a range of overhead values. One of the interesting results is that overheads under 5% have little effect on job performance.

We repeat the simulation with increasing fixed overheads (i.e. the performance of every job is uniformly reduced), up to 13% of CPU capacity (Figure 5). Increasing the overhead has little impact on system utilization: even with 13% overhead, utilization drops only 7%, from 0.89 to 0.83, and for overheads of 5% or less, utilization is not significantly affected. By contrast, average slowdown is more strongly affected by overhead, and at around 10%, the slowdown climbs steeply. This knee in the curve is an indicator of sustained system saturation [24], where the arrival rate of jobs exceeds the completion rate. The saturation point is important because it indicates that simulation results can be unreliable: the longer the simulation runs, the larger the slowdowns; there is no steady-state [16]. When the average load over time is considered (Figure 6), it can be seen that below 10% overhead, the load drops after the major spike at 350 days, and at 13% overhead, the load continues to climb, a clear indication of saturation.

Several studies [1], [5], [20], [21] of time-sharing restrict the multiprogramming level (MPL), which is the maximum allowable CPU load, to reduce both memory pressure and overhead. However, we assume an unlimited MPL because the CPU load is not that high in our simulation: it averages 9 over the full period, and does not exceed a weekly average of 20 during times of very high usage (for 0% overhead).

## V. Turnaround Scaling

Both time-sharing and batch scheduling make efficient use of the system, so utilization is similar and is largely determined by the rate of arrival of new jobs. However, the distribution of slowdowns is much more uniform for time-sharing and is not correlated with either job size or length. Consequently, there are regimes in which slowdowns are less under batch scheduling, and vice versa (Figure 7). As job length increases, batch slowdown improves and at around two hours, batch slowdown is lower than time-sharing slowdown, by up to 8×

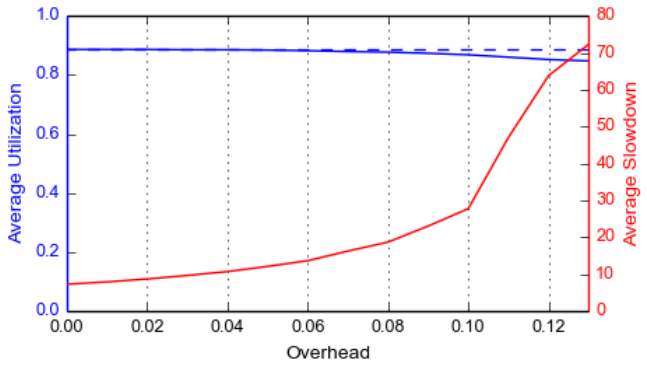


Fig. 5. Impact of overhead on time-sharing. The average utilization is the mean across all simulation timesteps and the average slowdown is the mean across all jobs. For comparison, the dashed line shows the average utilization for batch scheduling, which was not subject to any overhead. Slowdown is not significantly affected until around 5% overhead, and shows an abrupt increase at 10%.

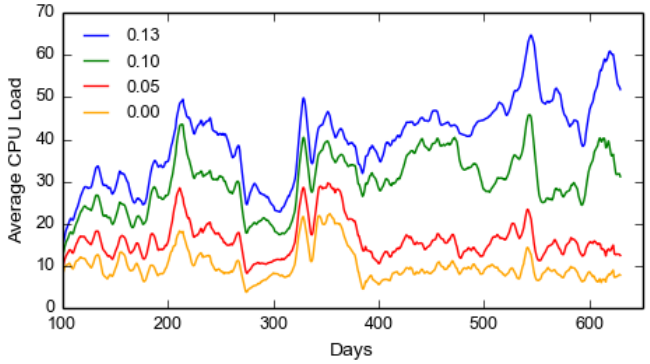


Fig. 6. Average CPU load over time for simulated time-sharing at various levels of overhead, as indicated in the legend. Each point is the CPU load averaged over all nodes and over the previous week (moving average). The system saturates around 10% overhead.

for very long jobs. However, for short jobs. slowdowns are orders of magnitude less under time-sharing. Time-sharing also tends to do better as job size increases, with the cross-over point shifting from around two hours to around four.

These differences have interesting implications for the user experience on systems such as Edison. Typically, the most important performance metric for a user is turnaround time: the sooner users can obtain the results of their simulations or analyses, the faster they can make scientific progress. For this reason, users invest great effort to ensure that applications strong scale far beyond what is needed to satisfy minimum memory requirements. But, is the effort actually worth it? Another way to look at this question is to ask: for a strong scaling application, what is the best concurrency to run at on a highly loaded system in order to minimize turnaround?

To investigate this question, we considered five applications from the APEX benchmark suite [25], and one simulated application ("perfect") that strong scales linearly. For the APEX applications, scaling data was collected from runs on Edison at various concurrencies. All of these applications

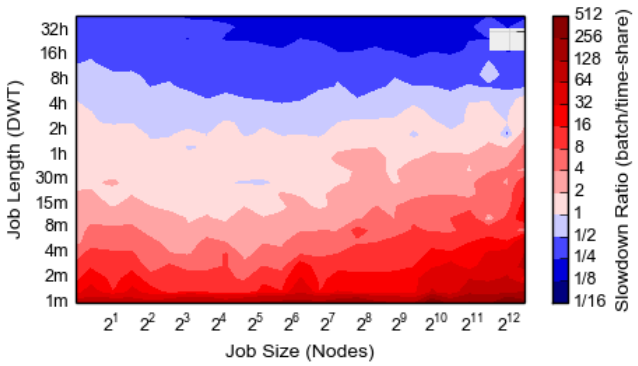


Fig. 7. Average slowdown ratio between batch scheduling and time-sharing, binned by size and length of job. Each point is colored according to the average slowdown of jobs with the batch scheduler, divided by the average slowdown of jobs with the time-sharing scheduler. Hence, red means time-sharing has better slowdowns, while blue means batch has better slowdowns. Batch tends to perform better than time-sharing for longer running jobs (over two hours).

strong scale to at least 960 nodes, with at least 49% efficiency.

Due to time and cost constraints, the set of (DWT, nodes) data points collected for each application was limited, and the applications were not run at the lowest possible concurrency. Since our goal is to determine how the turnaround for these applications is affected by a wide range of job sizes and lengths, we linearly interpolate between the measured data points, from the smallest size (determined by memory requirements) up to the maximum concurrency run.

We injected the test jobs into the simulation in multiple runs, at random points within the trace, and measured the average turnaround times. We also computed the turnaround times using the histograms of slowdowns (e.g. Figure 4), by taking the average slowdown of the bin for each (DWT, nodes) data point, and multiplying that by the DWT. These two approaches give very similar results; however we could not run a statistically significant sample of the test jobs on the real system, and so we report only the turnaround values computed from the histograms.

For the test applications, turnaround scales well with time-sharing, but not with batch scheduling (Figure 8). Under time-sharing, slowdown does not increase with a decrease in job length, and so increasing the job size results in a decreased job length, which decreases turnaround time. By contrast, with batch scheduling the bias we have observed against jobs with shorter DWT and more nodes results in an increasing turnaround precisely as jobs strong scale (increasing nodes and decreasing DWT). Consequently, at maximum size, the turnaround is usually orders of magnitude worse for batch scheduling than time-sharing. In some cases, when the minimum node count is high (GTC, HIPMER-WET), there is no concurrency at which the application can be run where the turnaround under batch scheduling will be even comparable to that under time-sharing.

Batch scheduling systems often attempt to address the problem of poor turnaround for large jobs by boosting their

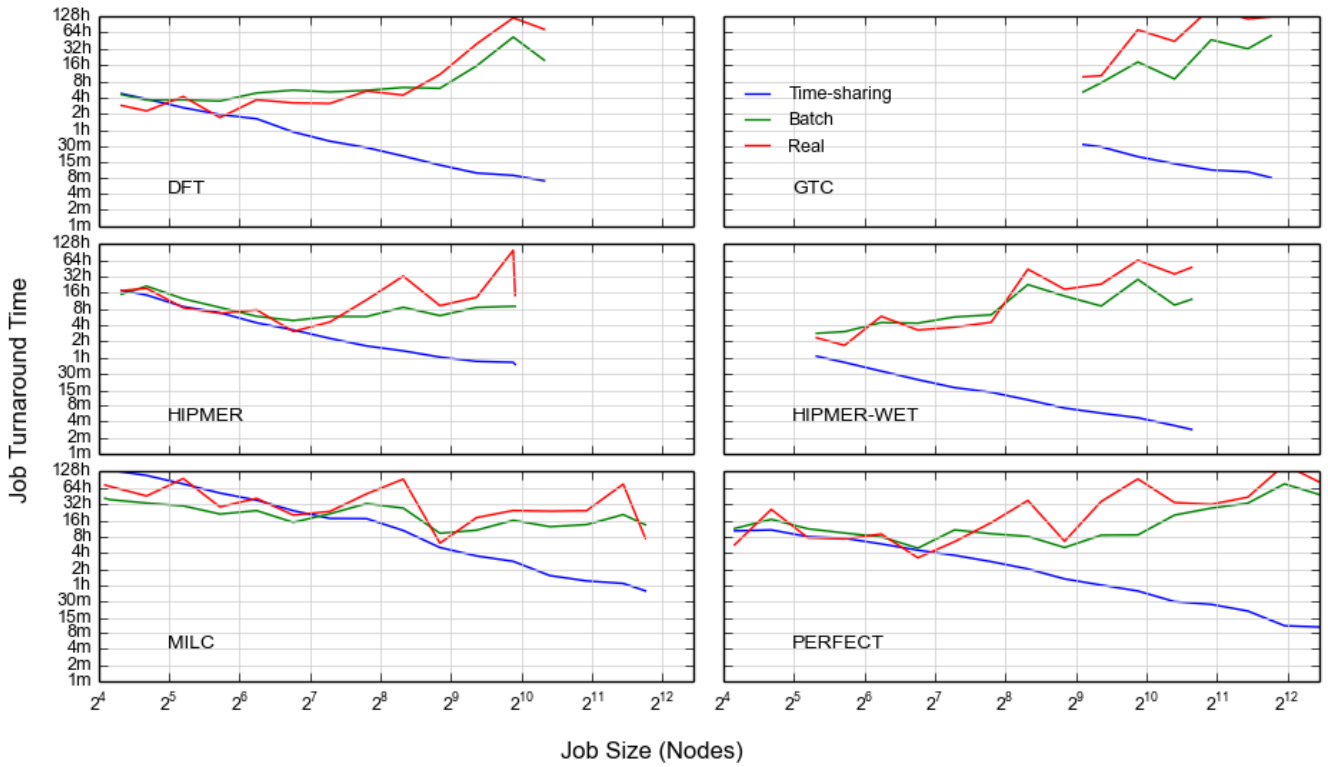


Fig. 8. Turnaround scaling for six strong scaling applications. Turnaround scales under time-sharing, but not under batch scheduling. Simulated batch scheduling gives very similar results to the Edison scheduler.

priority (the Edison scheduler boosts jobs with more than 682 nodes). However, the impacts are marginal, and do not significantly change the turnaround scaling, as can be seen from the fact that the simulated batch scheduling and real batch scheduling produce very similar results. Furthermore, even the larger boosts given to premium jobs are not adequate to change the pattern: premium jobs may have an average turnaround of double that of the normal jobs, but that is not sufficient to overcome the orders of magnitude skew for large, short jobs induced by batch scheduling.

## VI. Simulating Noise

Future HPC systems (up to exascale) are expected to be much more noisy than today's petascale systems [26]. This will require new approaches to avoid the serious performance loss that SPMD applications suffer in the presence of noise. One possibility is to use time-sharing instead of batch scheduling. In a heavily used system such as Edison, applications scheduled with time-sharing are not as susceptible to noise as batch scheduled applications.

We demonstrate this point with a simple noise model: every simulation timestep, each node has a probability $p = 0.0001$ of running at reduced speed for the duration of the timestep. The reduced speed is chosen uniformly at random from the interval $[0.5, 1.0]$. This can be seen as a simple model of noise generated by system services. In our simulation of 5576 nodes there will be on average five or six nodes delayed at every timestep. This gives a net processor noise of $0.025\%$, much less than the $2.5\%$ commonly used in prior studies [27].

With noise, the slowdown under time-sharing is less than under batch scheduling almost everywhere (Figure 9). By contrast, without noise batch scheduling is better for jobs of DWT greater than two to four hours, depending on job size (Figure 7). Such differences in the impact of noise are to be expected. Under batch scheduling, if an application slows down on a single node there are no other applications to utilize the excess cycles on the application's other nodes. By contrast, with time-sharing, there is a strong possibility that another application not affected by noise can utilize the spare cycles.

Under batch scheduling noise causes an increase in average slowdown across all jobs from 137 to 1009. However, the average reflects a large increase in the tail of the distribution, and for most jobs the impact is not as extreme, e.g., the 75th percentile increases from 13 to 22. The tail increase is skewed towards larger jobs, because, given that a SPMD application runs at the speed of its slowest node, the larger the job, the greater the chance that one of its nodes will be degraded.

We see the same effect in time-sharing, where larger jobs experience higher slowdowns, up to an order of magnitude worse than the smaller jobs. However, this skew is still not as strong as batch scheduling, where the difference in slowdowns between small and large jobs is several orders of magnitude. With time-sharing, there is far less impact on the tail of the distribution, and consequently the average slowdown only

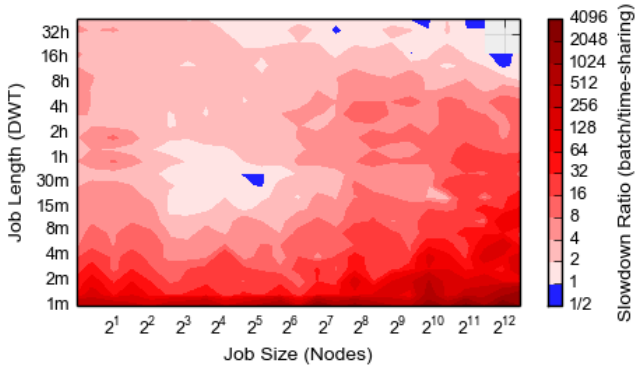


Fig. 9. Average slowdown ratio between time-sharing and batch scheduling in the presence of noise, binned by size and length of job. Each point is colored according to the average slowdown of jobs with the batch scheduler, divided by the average slowdown of jobs with time-sharing. Red indicates that slowdown is lower under time-sharing, and blue indicates that it is lower under batch.

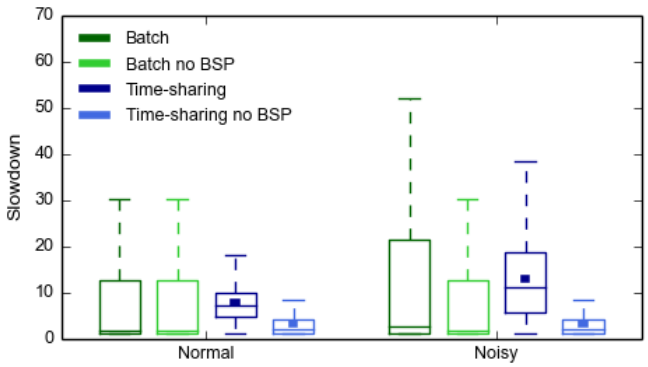


Fig. 10. Slowdown distributions for simulated schedulers. Each box gives the inter-quartile range (IQR), the horizontal line in the box is the median, the whiskers show $\mathbf{1.5}\times$IQR, and the blocks are the average. The averages for batch scheduling fall outside the maximum y-value (137 without noise, 1009 with noise). Because the distributions are heavy-tailed, the outliers are omitted.

increases from 8 to 13.

## VII. Programming Model Impact

We have assumed that all applications use the SPMD model, i.e. they make progress at the speed of the slowest node. Under time-sharing, an application will not be able use excess cycles on one node if it is proceeding more slowly on another due to higher CPU load (or noise). Relaxing the SPMD assumption enables us to explore the consequences of alternative, asynchronous programming models. Although asynchronous programming models are not currently commonly used in HPC, they will likely become essential for exascale because of their potential to perform better in unpredictable, heterogeneous environments at massive concurrencies [28].

Relaxing the SPMD assumption shifts the results significantly in favor of time-sharing, with slowdowns under time-sharing being lower than under batch for all but the longest running jobs (those with DWT $\geq$ 16 hours). By contrast, with the SPMD assumption, time-sharing is only better than batch for jobs with DWT under two to four hours (Figure 7). This relative improvement is a consequence of the fact that non-SPMD applications can utilize excess cycles in time-sharing, which cannot happen in batch scheduling.

Figure 10 summarizes the impact of both noise and programming models. Non-SPMD jobs are not significantly affected by the low level of noise we simulated, when scheduled with either batch or time-sharing. The average slowdowns for batch scheduling are too large to be shown on the plot, indicating heavy-tailed distributions. By contrast, the averages for time-sharing are close to the medians, indicating distributions with much lower variance.

## VIII. Discussion

The simulation results we presented are based on the actual job traces of Edison usage. This has the benefit of giving a realistic pattern of usage, one that varies over time, incorporating burstiness and other irregularities. However, it limits the results to one particular supercomputer, with one specific usage pattern, and the system load or other parameters cannot be adjusted. An alternative is to model the distribution of jobs (in terms of arrival rates, job size, length and memory usage), and use jobs generated at random from the distribution. Although validation becomes a challenge, the modeling approach gives the flexibility to explore different parameters, such as system load, and the ability to simulate larger systems [29]. This is a topic for future work; in particular, we would like to see how the results change as the system scales up to exascale.

The issue of memory pressure in time-sharing is an important one. We simulated admission control, which requires a priori values for peak memory usage for every job. An alternative is to allow jobs to oversubscribe memory. However, current supercomputers typically have no node-attached storage, and no ability to swap out pages to disk. One proposed solution to this problem is using virtual machines to suspend a job to disk [4], [5]. This approach is expensive because it relies on access to remote storage, and even if the suspension events are relatively infrequent (in our simulation, the vast majority of jobs are admitted immediately), the system-wide cost could be prohibitive. Fortunately, this problem is likely to disappear with the increasing use of NVRAM for local storage; for example, the latest NERSC supercomputer, Cori, incorporates burst buffers, which utilize NVRAM linked to every compute node [30]. We believe this will enable the operating system to multiplex applications efficiently, and deal with oversubscribed memory by efficiently paging out (in traditional operating system style) to the burst buffers.

We have demonstrated how noise can impact job performance, but our simulation is limited to one particular noise profile. Ferreira et al. [27] explore different noise profiles, including high frequency noise representative of timer interrupts and low frequency, high duration noise, representative of kernel daemon interference. We are not able to simulate noise at this level because our timestep is 10s, and even the low frequency noise investigated by Ferreira et al. is 10Hz. A topic

for future work is incorporating more realistic models of noise and system performance variation, especially as projected for future exascale systems.

The simulations with the non-SPMD jobs have interesting implications for the impact of programming models on future systems. We have shown that relaxing the SPMD assumption with time-sharing not only improves the overall system utilization, but also improves the experience for users by reducing the slowdown. This is particularly beneficial in noisy environments, but even in predictable environments the improvement is significant, with average slowdown dropping from 7.9 to 3.3. This improvement is enough that even if non-SPMD applications are somewhat less efficient, that is unlikely to compromise the overall usage or user experience, even with a highly predictable, noise-free system. In other words, the optimum approach for a capacity system may not be the best approach for a capability system.

## IX. Related Work

In their seminal paper [7], Feitelson and Rudolph showed that gang scheduling only has performance benefits for fine-grain BSP applications that use busy-wait for synchronization. However, gang scheduling suffers from a fragmentation problem because of the requirement for coordinating time slicing across nodes. Consequently, other research has investigated how to achieve the performance benefits of gang scheduling without incurring fragmentation or having to deal with implementation complexity and overhead. Implicit co-scheduling [12] relies on two-phase "spin-blocks" to enable local schedulers to operate independently and the synchronization between different threads emerges naturally when threads block on IO. Flexible co-scheduling [21], [31] monitors the communication activity of jobs and uses gang scheduling only for those jobs that are likely to need it. Using blocking synchronization on a Linux cluster combined with local scheduling outperforms gang scheduling on a variety of parallel benchmarks [22]. These results are supported by a detailed analysis [32] of the various dynamic co-scheduling approaches. We do not simulate gang scheduling, but instead assume that BSP applications can achieve efficient performance through one of these alternative scheduling approaches.

The earliest study [1] comparing space-sharing with time-sharing showed that gang scheduling could outperform batch scheduling and result in increased system utilization for production workloads. Further research [2] comparing gang scheduling with batch scheduling has shown that augmenting gang scheduling with backfill provides additional performance benefits under time-sharing. Another simulation study [3] comparing gang scheduling to batch scheduling also shows performance benefits for time-sharing in certain cases.

Our work bears many similarities to these comparative studies; however, there are a number of important differences. First, we do not assume gang scheduling. Second, we use a large modern jobs trace, collected over a 20 month period on a modern (circa 2015) petascale supercomputer. Third, we explore the impact of both job size and length on the slowdown distributions in detail, and consequently discover explicit regions of the parameter space where time-sharing does better than batch scheduling, and vice versa. This detailed analysis enables us to investigate turnaround scaling, an aspect that was not considered in previous studies. Finally, we consider the impacts of both noise and programming models, aspects which are largely omitted from previous studies.

The closest related work to the results presented here is Fractional Resource Scheduling [4], [5] (FCS), which assumes that virtual machine technology can be leveraged to provide uncoordinated time-sharing without the limitations of gang scheduling. Like our work, the FCS simulations also assume BSP applications and use admission control for memory constraints. Their goal is to minimize the maximum slowdown (worst case), and for this purpose they find that time-sharing is much better than batch scheduling, which agrees with our results. However, considering only the maximum slowdown gives a limited picture of the trade-offs between schedulers, because slowdowns are heavy-tailed under batch scheduling, making for much higher maximums, even though there are cases when batch is better on average (e.g. long jobs). They also do not investigate turnaround scaling, noise or alternative programming models, and use a much smaller dataset (200,000 jobs) from a much smaller system (128 nodes).

An alternative approach to scheduling, also inspired by the advent of virtualization, is job folding [33]. The idea is to multiplex virtual processors onto real processors to improve the flexibility of scheduling by avoiding the geometrical constraints of pure space-sharing. They show improvements over FCFS with backfill in limited simulations (1024 processors and 10,000 jobs). Although the approach is interesting, their job model is a simple Poisson, which is likely unrealistic [16], and they do not consider memory constraints.

## X. Conclusions

This paper presents a simulation study that explores the impact of time-sharing on Edison, a modern petascale supercomputer. We showed that although batch scheduling and time-share scheduling result in similar overall system utilization, the distribution of job slowdowns is quite different. Batch scheduling results in a heavy-tailed distribution, where shorter-running, smaller jobs have much larger slowdowns, whereas time-sharing produces a relatively uniform distribution of slowdowns, regardless of job size or length. Consequently, time-sharing is better than batch scheduling (in terms of slowdown) for short jobs (under two to four hours), with an improvement of over $500\times$ for the shortest jobs; by contrast, under batch scheduling, long jobs can see an improvement of up to $8\times$ over time-sharing.

A consequence of the skewed distribution of slowdowns under batch scheduling is that turnaround does not scale. We showed this by determining the expected turnaround for six applications that strong scale, with both batch scheduling and time-sharing. As the applications strong scale, the job size increases and the length decreases, which is exactly the region

in which batch scheduled jobs experience higher slowdowns. By contrast, with time-sharing, the uniform distribution of slowdowns results in strong scaling of the job turnaround, and by the time maximum scale is reached, the turnaround for the applications under time-sharing is orders of magnitude better than under batch scheduling (e.g. 4 minutes vs 8 hours).

Our study also showed that BSP applications are susceptible to very low levels of noise (0.025% net processor noise), whether the system was using batch scheduling or time-sharing. However, the effect on time-shared jobs is lower, because excess CPU cycles from a job slowed down by noise can be used by other, unaffected jobs. Consequently, in the presence of noise time-sharing is better than batch scheduling in terms of slowdown, almost regardless of job size or length. In addition, we showed that when the BSP assumption is relaxed, the slowdown under time-sharing improves (the average is halved), whereas with batch scheduling it makes no difference. In both cases, non-BSP applications are unaffected by noise.

## XI. Acknowledgments

This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research.

## References

[1] D. G. Feitelson and M. A. Jettee, "Improved utilization and responsiveness with gang scheduling," in *Job Scheduling Strategies for Parallel Processing*. Springer, 1997, pp. 238–261.

[2] Y. Zhang, H. Franke, J. Moreira, and A. Sivasubramaniam, "Improving parallel job scheduling by combining gang scheduling and backfilling techniques," in *IEEE IPDPS*, 2000, pp. 133–142.

[3] ——, "A comparative analysis of space-and time-sharing techniques for parallel job scheduling in large scale parallel systems," *IEEE Transactions on Parallel and Distributed Systems*, 2002.

[4] M. Stillwell, F. Vivien, and H. Casanova, "Dynamic fractional resource scheduling for hpc workloads," in *IEEE IPDPS*, 2010, pp. 1–12.

[5] H. Casanova, M. Stillwell, and F. Vivien, "Dynamic fractional resource scheduling vs. batch scheduling," *arXiv preprint arXiv:1106.4985*, 2011.

[6] J. K. Ousterhout, "Scheduling techniques for concurrent systems." in *ICDCS*, vol. 82, 1982, pp. 22–30.

[7] D. G. Feitelson and L. Rudolph, "Gang scheduling performance benefits for fine-grain synchronization," *Journal of Parallel and Distributed Computing*, vol. 16, no. 4, pp. 306–318, 1992.

[8] A. Verma, L. Pedrosa, M. R. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at Google with Borg," in *EuroSys*, Bordeaux, France, 2015.

[9] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica, "Mesos: A platform for fine-grained resource sharing in the data center." in *USENIX NSDI*, 2011, pp. 22–22.

[10] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler, "Apache Hadoop YARN: Yet another resource negotiator," in *SOCC*, 2013.

[11] NERSC, "Edison," https://www.nersc.gov/users/computational-systems/edison, 2016 (accessed January 2016).

[12] A. C. Arpaci-Dusseau, D. E. Culler, and A. M. Mainwaring, "Scheduling with implicit information in distributed systems," *ACM SIGMETRICS Performance Evaluation Review*, vol. 26, no. 1, pp. 233–243, 1998.

[13] D. G. Feitelson, L. Rudolph, U. Schwiegelshohn, K. C. Sevcik, and P. Wong, "Theory and practice in parallel job scheduling," in *Job scheduling strategies for parallel processing*. Springer, 1997, pp. 1–34.

[14] TACC, "Tacc," http://www.tacc.utexas.edu, 2016 (accessed January 2016).

[15] NCSA, "Ncsa," http://www.ncsa.illinois.edu, 2016 (accessed January 2016).

[16] E. Frachtenberg and D. G. Feitelson, "Pitfalls in parallel job scheduling evaluation," in *Job Scheduling Strategies for Parallel Processing*. Springer, 2005, pp. 257–282.

[17] E. Frachtenberg, D. G. Feitelson, J. Fernandez, and F. Petrini, "Parallel job scheduling under dynamic workloads," in *Job Scheduling Strategies for Parallel Processing*. Springer, 2003, pp. 208–227.

[18] D. G. Feitelson and L. Rudolph, "Metrics and benchmarking for parallel job scheduling," in *Job Scheduling Strategies for Parallel Processing*. Springer, 1998, pp. 1–24.

[19] M. G. Kendall, "Rank correlation methods." 1948.

[20] A. Batat and D. G. Feitelson, "Gang scheduling with memory considerations," in *IEEE IPDPS*, 2000, p. 109.

[21] E. Frachtenberg, D. G. Feitelson, F. Petrini, and J. Fernandez, "Flexible coscheduling: Mitigating load imbalance and improving utilization of heterogeneous resources," in *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*. IEEE, 2003, pp. 10–pp.

[22] P. Strazdins and J. Uhlmann, "A comparison of local and gang scheduling on a beowulf cluster," in *IEEE Int'l Conference on Cluster Computing*, 2004, pp. 55–62.

[23] C. Iancu, S. Hofmeyr, F. Blagojević, and Y. Zheng, "Oversubscription on multicore processors," in *IEEE IPDPS*, 2010, pp. 1–11.

[24] L. Rudolph and P. H. Smith, "Valuation of ultra-scale computing systems," in *Job Scheduling Strategies for Parallel Processing*. Springer, 2000, pp. 39–55.

[25] NERSC, "APEX benchmarks," http://www.nersc.gov/research-and-development/apex/apex-benchmarks, 2016 (accessed January 2016).

[26] D. L. Brown, P. Messina, P. Beckman, D. Keyes, J. Vetter, M. Anitescu, J. Bell, R. Brightwell, B. Chamberlain, D. Estep *et al.*, "Cross cutting technologies for computing at the exascale," US Department of Energy (DOE) Office of Advanced Scientific Computing Research and the National Nuclear Security Administration, Tech. Rep., 2010.

[27] K. B. Ferreira, P. Bridges, and R. Brightwell, "Characterizing application sensitivity to os interference using kernel-level noise injection," in *ACM/IEEE Conference on Supercomputing*, 2008, p. 19.

[28] S. Amarasinghe, M. Hall, R. Lethin, K. Pingali, D. Quinlan, V. Sarkar, J. Shalf, R. Lucas, K. Yelick, P. Balaji *et al.*, "Exascale programming challenges," in *Report of the 2011 Workshop on Exascale Programming Challenges, Marina del Rey*, 2011.

[29] C. Ernemann, B. Song, and R. Yahyapour, "Scaling of workload traces," in *Job Scheduling Strategies for Parallel Processing*. Springer, 2003, pp. 166–182.

[30] NERSC, "Cori burst-buffers," https://www.nersc.gov/users/computational-systems/cori/burst-buffer, 2016 (accessed January 2016).

[31] E. Frachtenberg, D. G. Feitelson, F. Petrini, and J. Fernandez, "Adaptive parallel job scheduling with flexible coscheduling," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 16, no. 11, pp. 1066–1077, 2005.

[32] M. S. Squillante, Y. Zhang, A. Sivasubramaniam, N. Gautam, H. Franke, and J. Moreira, "Modeling and analysis of dynamic coscheduling in parallel and distributed environments," *ACM SIGMETRICS Performance Evaluation Review*, vol. 30, no. 1, pp. 43–54, 2002.

[33] J.-M. Nicod, L. Philippe, V. Rehn-Sonigo, and L. Toch, "Using virtualization and job folding for batch scheduling," in *ISPDC*, 2011, pp. 33–40.