A Better Way of Scheduling Jobs on HPC Systems: *Simultaneous* Fair-Share

Craig P Steffen
csteffen@ncsa.illinois.edu
NCSA: University of Illinois

ABSTRACT

Typical HPC job scheduler software determines scheduling order by a linear sum of weighted priority terms. When a system has a rich mix of job types, this makes it difficult to maintain good productivity across diverse user groups. Currently-implemented fair-share algorithms tweak priority calculations based on *past* job handling by modifying priority, but don't fully solve problems of queue-stuffing and various classes of under-served job types, because of coupling between different terms of the linear calculated priority .

This paper proposes a new scheme of scheduling jobs on an HPC system called "Simultaneous Fair-share" (or "SFS") that works by considering the jobs already committed to run in a given time slice and adjusting which jobs are selected to run accordingly. This allows high-throughput collaborations to get lots of jobs run, but avoids the problems of some groups starving out others due to job characteristics, all while keeping system administrators from having to directly manage job schedules. This paper presents Simultaneous Fair-share in detail, with examples, and shows testing results using a job throughput and scheduler simulation.

CCS CONCEPTS

• **Theory of computation** → *Scheduling algorithms*; • **Computing methodologies** → *Planning under uncertainty*.

KEYWORDS

scheduling, fairshare, fair-share, fairness, scheduling throughput

ACM Reference Format:
Craig P Steffen. 2019. A Better Way of Scheduling Jobs on HPC Systems: *Simultaneous* Fair-Share. In *SIGHPC Systems Professionals Workshop, HPC-SYSPROS19, November 22, 2019, Denver, CO.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Scheduling software in HPC systems typically prioritizes jobs using a linear sum of weighted terms which are calculated based on the job's characteristics[2][7]. Such a linear priority scheme is simple to configure and simple to understand, but that simplicity comes with the price that different scheduling considerations are consequently coupled, because their effect is to raise or lower the priority of the job, which is a single number. As the mix of submitted jobs changes, so does how jobs flow through the system changes, and administrators are constantly faced with the task of shifting priorities to make sure jobs are flowing through the system in a reasonable fashion.

The proposed Simultaneous Fair-Share ("SFS") job scheduling algorithm solves that issue. SFS separates fair-share considerations from other linear priority algorithm factors. Administrators can weigh job priority according to what jobs *should* be weighted while the SFS algorithm itself makes sure all groups of users are getting jobs through. This has several benefits. It increases user satisfaction because the system is perceived to be more responsive, especially for debug or testing jobs. It increases the throughput of chained jobs, such as a physical simulation that must operate multiple sessions on a single evolved physical simulation that is handed from job to job. Most importantly, it promotes fair job throughput as part of the scheduling algorithm rather than as a consequence of priority weighting.

2 LINEAR SCHEDULING OVERVIEW

In default versions of popular schedulers, like Moab and Slurm, jobs are selected to be scheduled to run in order of "priority", which is a calculation performed for each job at the beginning of every scheduling pass. Numerical priority for a job is the sum of terms of characteristics of that job, each multiplied by a numerical weight assigned as part of the configuration of the scheduler software assigned by the system administrators. The weights are assigned to emphasise or de-emphasise certain characteristics of jobs that are considered desirable to run on the given computational resource.

2.1 Priority by job size

A common weighing factor is to have the numerical priority for a queued job increased proportionally to the number of resources that a job requests (the "size", or the number of nodes in a job). This prioritizes large jobs over small jobs, counteracting the ease of scheduling small jobs. Node size is one of the important factors used in operational scheduling with Moab on Blue Waters[6]. Blue Waters was promoted as a capability systems, so our priority of getting very large capability jobs through the system quickly has been reflected in a strong weight on priority due to job size.

2.2 Priority by job age

Another common adjustment factor is how much a long wall time increases the job's priority. This factor counter-acts the fact that shorter jobs are easier to schedule than long ones. All of the other adjustment factors must be balanced against factors that further increase a job's numerical priority the longer that it has been in the queue eligible to run. This factor counteracts the other priority factors and works to make sure that jobs don't get too stale waiting in the queue. As a job ages, its priority goes up with time and its priority is eventually greater than that of other, newer jobs that would have otherwise gone first.

These are just a few common competing priorities that typical scheduling software allows admins to adjust. They're adjusted to allow different types of jobs to flow through the scheduling system and be run on the machine. They will need to be adjusted when the characteristic workload changes significantly. They can stay the same if the workload stays roughly the same, and users (and groups of users) submit jobs to the system at a steady rate that is reasonably close to the rate that the system can run the jobs.

2.3 Priority Calculation

The stock configuration of Slurm and Moab calculate the instantaneous priority of a job as a linear sum of terms, as follows: $P_i = A \cdot a_i + B \cdot b_i \ldots$ The term weights A and B are weighting factors set by the configuration of the scheduling software, and are typically constant for all jobs. The other factors in the terms (a_i, b_i, and so on) are calculated on a per-job basis based on the characteristics of the job, and perhaps other outside information such as the eligibility of the user or the job.

Linear priority calculation focuses on the characteristics of the job, not the characteristics of the user. Jobs of certain characteristics (say, larger jobs) are going to be prioritized over jobs with other characteristics that aren't weighed as much. Due to random circumstances of timing and other consistencies in jobs submitted by groups of users, jobs for one group may tend to dominate over jobs by other groups. Fair-share strategies are employed to combat this tendency in schedulers.

2.4 History-based Fair-share

Both Moab and Slurm have fair-share mechanisms[1][11] available. These mechanisms add a negative term to the priority sum that lowers the priority of a user's (or user group's) jobs based on resource usage of jobs they have run in a recent time window. Because that group exceeded some administratively defined threshold of usage, that group's job's priority is lowered so their jobs are less likely to run than users' or groups' that hadn't been running as much.

3 PROBLEMS WITH LINEAR PRIORITY SCHEDULING

Linear priority scheduling inherently couples all of the priority factors and so makes it difficult to decouple Fair-share considerations from other job characteristics. This is why a scheme that separates out Fair-share considerations from other priority calculations is vital to good job flow through a system.

3.1 Queue Stuffing

Jobs flow through an HPC system reasonably as long as most users submit jobs at a slow steady rate. But frequently users running on a system don't follow that pattern. Sometimes a group on a system will have all of their input data ready when their allocation opens, and so they can submit jobs that represent a large chunk of their total allocation in a sort time. If an allocation is large enough, they could submit enough workload at once that the system could take weeks to run all of their jobs. If the scheduler is configured to increase job priority due to age, then just by job plurality alone, this large group of jobs will dominate the queue until they have been depleted, making it hard for other jobs to run.

This phenomenon is called "queue-stuffing". It's a great way for users to submit jobs. If all users did this, then the scheduler would be able to weigh what jobs to run for the maximum total efficiency of the system over a long period of time. However, other groups, who submit fewer jobs at first and start submitting jobs later in the year, might only get jobs through after a very long latency. This makes it difficult for those other allocations to get their jobs tuned and tested before they need to get their main production workflow running. It's also frustrating for a smaller or newer allocation to not be able to get anything through for weeks because larger, more well-established allocations are completely dominating the queue. Simultaneous Fair-share deals with this exact problem.

Let us make one very specific point about the phenomenon of queue-stuffing: it is a natural consequence of a large system with large, well-prepared user groups. A well-organized collaboration with a lot of input data to process may well squeeze other user groups who aren't immediately ready to deploy their entire production workload. Queue-stuffing is not a user problem–is a failure of the scheduler infrastructure to balance the use of the system among users of different input patterns. Schedulers should be able to usher through user jobs reasonably and efficiently even when user groups significantly differ in their submission techniques, and SFS does this.

3.2 Problems with History-based Fair-share

Fair-share[1] is a mechanism in popular schedulers[1][11] that attempts to mitigate the effects of queue stuffing. However, it is only a partial solution. History-based fair-share in those scheduler applications lowers the linear numerical priority of queued jobs whose recent finished user's jobs exceed a target occupancy of job resources. The problem is they're only sensitive to jobs that have been completed and logged by the rolling fair-share calculation. Scheduling of the jobs of an allocation will only be effected by jobs that have finished, not that are currently running. If a group submits a bunch of jobs quickly, they could potentially fill the entire system with their jobs. History-based allocation wouldn't effect future queued jobs until those jobs had all finished and the fair-share calculation registered them. Because jobs tend to get reservations because they're high in the queue, by the time that group's priority has been lowered, some of that group's *next* jobs may already be scheduled to run.

In addition to the generic problems with history-based scheduling, Moab specifically has characteristics that make this problem worse. Sometimes it takes a couple of scheduler iterations in Moab for the fair-share results to propagate and the results felt in the priority calculation. So one group can fill the machine with jobs, and fair-share won't effect the jobs at the top of the queue currently waiting to be scheduled. Those waiting jobs will be scheduled and

[1]Fair-share as commonly implemented in Moab and Slurm is based on job history. I'm defining the term "history-based fair-share" in this paper, and referring to traditional fair-share as that, to contrast it to "Simultaneous Fair-share", to make it clear which of the two I'm referring to.

have reservations when the first jobs finished. Because of the latency in the fair-share logging, the second wave of jobs will be scheduled and start running before the first wave of jobs have been entered into the fair-share job log. By the time the first wave has been entered, the second wave of jobs have already started. If a group has stuffed the queue, then they can sometimes get one wave of jobs, and a second wave of jobs before fair-share effects that group's priority values. In the recent period of Blue Waters operations, this would result in another group having to wait at least 4 days (two times the maximum allowed job duration) between submitting a first job and any chance of having it run.

3.3 Under-served Job Classes

Depending on the input data requirements and the application workflow, there are at least two different job types that users tend to submit. One type of job operates on a discrete piece of data, and that data is known at submit time. This paper calls these "bulk" jobs. The bulk jobs can be run in any order, many of them at a time. Many well-established collaborations have workflows that look like this. Physics collaborations, for instance, that have large ensemble calculations to run have large collections independent data sets that need to be processed. Those jobs can be run in any order, at any time. This kind of workflow, because it can be submitted all at once, tends to dominate queue structure and can very easily, through no fault of the people submitting it, squeeze other, smaller jobs out of contention for running.

Another common type of job is a "serial" job. Those kinds of jobs are typical of a long physical simulation, the results of one job become the input to the next job. Only one job can usefully run at a time, and the next job in the trail must wait until the first is finished before being eligible to run.

With the current versions of Moab and Slurm, serial jobs have problems getting through the queue when there are a lot of bulk jobs in the queue at the same time. It's difficult to configure linear job parameters such that serial jobs chain together efficiently. There are mechanisms within the scheduling software to help serial jobs work better, but configuring those in scheduling software opens up that effect for exploitation by users to get their jobs to run artificially fast.

In historical system running on Blue Waters, serial jobs tend to be under-served because of the scheduler configuration conflicts listed above. Jobs in the range of 256 to 1500 nodes tend to be served poorly as well. They sometimes take 2 and 3 days to run, even when the system load is fairly light. Worse, serial jobs in the 256 to 1500 node range are especially poorly served. These jobs' performance on Blue Waters were a large portion of the motivation for this work of trying to find a better way to schedule jobs on a large system, taking fair-share into account at a fundamental level, rather than patching it in with historically-based fair-share.

3.4 Linear Priority Scheduling Promotes Gaming The Scheduler

Linear priority scheduling, by its nature, selects jobs that have certain desired characteristics. With current versions of Moab and Slurm and other available schedulers, The system administrators must adjust the priority weighting of certain job characteristics in order to make sure that all groups are being represented in the job mix that the system is running. That also means that users are incentivized to construct their jobs to match what the scheduler currently "wants". Because linear priority scheduling combines all of the job characteristics to a single number, adjusting the priority weightings to match the current job workload is inescapable. As new groups move on and off the system, this becomes a resource drain on the administrators to keep abreast of changes in user behavior and allocation mixes.

3.5 Managing the Scheduler with Reservations

With traditional scheduling systems, the admins must adjust priorities to try to make sure all types and sources of jobs are running. If adjusting parameter weights isn't sufficient, then job groups that are being under-served can be promoted by giving them standing reservations on the system. This is very effective for that group of jobs, but keeping track of reservations requires accounting on both the sysadmins' and the user's parts. It also means the admins have to in turn remove reservations to let large jobs through, otherwise they won't have room to run at all. Having manually-managed static user reservations has become common in the 2018-2019 operational year of Blue Waters, Simultaneous Fair-share directly addresses this specific problem and and all the general problems with linear priority weighting.

4 SIMULTANEOUS FAIR-SHARE

Simultaneous Fair-share is a new way of ordering and running jobs in a scheduler system that deals with the shortcomings of the above scheduling compromises in a way that makes scheduling more efficient and also requiring far less effort from the system administrators to keep things flowing well.

This proposed definition of the Simultaneous Fair-share scheduling strategy assumes an underlying scheduling system such as Moab or Torque as discussed above. There's a pool of jobs, which have linear numerical priorities assigned to roughly balance out their running orders. However, Simultaneous Fair-share replaces scheduling logic based purely on running the highest numerical priority first with a multi-pass loop that gates when jobs are eligible to be run at a given time. What jobs get scheduled at a given time is a function of what other jobs are running, what allocation they are under. This scheduling logic knows what jobs are attached to what allocations, and more information specific to that allocation as outlined below.

4.1 Ideal Throughput and Target Throughput

Simultaneous Fair-share is based on the idea that every allocation on a system has an "ideal occupancy" (sometimes also referred to as "ideal throughput"). That's a size of job that if they had one job of that size continuously running for the entire calendar period of their allocation, they would exactly use their allocation up just as the allocation period expired. This ideal occupancy is calculated by dividing the total amount of their allocation (which is in units of resource size times time, for instance CPU*hours) and then dividing by the length of the allocation. So for instance, if allocation "Alice" has one million CPU*hours on a system and their allocation goes for one year, then their ideal occupancy (or throughput) T is:

$$T^{ideal}_{Alice} = \frac{1e6\ CPU * hr}{1yr} \times \frac{1yr}{365dy} \times \frac{1dy}{24hr} \approx 114\ nodes \quad (1)$$

If the Alice collaboration configured their application to use exactly 114 nodes, and the application started running when their allocation was awarded, and it ran the entire year, their allocation amount would be exhausted exactly when their allocation calendar time expired. The idea is that if all the allocations on a system used their allocation at their ideal occupancy rate, the system would be 100% utilized and every allocation would be able to get their science done in their alloted time.

In Simultaneous Fair-share scheduling scheme, ideal occupancy is an important illustrative concept, but it's not used directly. No allocation is prepared to run continuously. Operationally there has to be the assumption there is time for debugging, run preparation, and presumably sleep on the part of the people doing the work. So every allocation has a "target occupancy" defined. This is some factor larger than their ideal throughput. For the purposes of illustration and simplicity, this paper defines the Alice collaboration's target occupancy as double their ideal occupancy.

$$T^{target}_{Alice} = (2) \times T^{ideal}_{Alice} = 228\ nodes \quad (2)$$

Every allocation on a system has this target occupancy defined. It's static over time, it's only a function of the original awarded amount of use time, it doesn't change as the allocation is used. Typically each allocation's target occupancy would be the same factor above their ideal occupancy as other allocations, but that could be adjusted by the administrators on a case-by-case basis. The target occupancy for each allocation is used in the Simultaneous Fair-share job placement algorithm which is described in following sections.

4.2 Description of Simultaneous Fair-share Scheduling Process

When traditional schedulers prioritize jobs, they do so using the numerical priority values of the individual jobs, and place (or reserve space for them) in order of numerical priority and nothing else. All the scheduling considerations for each job are built into that one numerical value. That limits prioritization to a one-dimensional continuum and leaves out many subtleties of context and history of other jobs being run, which is why that scheme often has to be adjusted when the job load changes significantly.

Simultaneous fair-share still uses numerical priority for ordering jobs, but that's not the overriding consideration for job placement. Instead, Simultaneous Fair-share splits job placement decisions into several separate logical passes through the eligible job list. The job list that each pass uses is specific to that pass and can be dynamic within the pass.

4.2.1 Scheduling Pass 1. When there's an opportunity to schedule jobs to be run on the resource, SFS first builds its local first-pass job list out of the overall list of eligible jobs. To do this, SFS builds a list of all of the allocations on the system. For the time slice that needs to have jobs scheduled, it goes through jobs that are already running or scheduled to run at that time and totals the size (in resources, CPUs or nodes) of jobs running or scheduled to run at that time slice. Then the total for each allocation (within the time slice) is compared to that allocation's target occupancy. Allocations whose occupancy for that time slice exceed their target occupancy have their jobs removed from the local job list. Allocations whose occupancy are equal to or below their target have their jobs left on the job list.

Once the local job list is built for this pass and this time slice, then scheduling is done for that time slice from the remaining job list, ordered by numerical priority as in traditional scheduling. After each job is placed, the per-allocation occupancy list is updated to reflect the placed job. If the placed job causes an allocation now be above target occupancy for this time slice, the all remaining jobs for that allocation are removed from the local job list. This continues until either no more jobs from the local list can fit at this time, or else the local job list is empty.

4.2.2 Scheduling Pass 2. SFS then makes a second pass of scheduling for that same time slice, but this time using the full eligible job list, numerically prioritized as before. In the second pass, no consideration is made whether or not allocations are above their target occupancy.

5 EXPLICIT EXAMPLE OF SFS WORKING ON A SMALL SET OF JOBS

This is a simple example to illustrate the Simultaneous Fair-share process. We have a system with 1000 nodes. There are two allocations running on the system. The "*Alice*" collaboration and the "*Bob*" collaboration. Their target occupancies are $T^{target}_{Alice} = 288\ nodes$ and $T^{target}_{Bob} = 58\ nodes$. The Alice allocation runs jobs that are 200 nodes, the Bob allocation's jobs are 50 nodes each. Each allocation has submitted 10 jobs, Alice's jobs are $A, B, C, D, E, F, G, H, J, K$, and Bob's jobs are $M, N, P, Q, R, S, U, V, W, X$. There's a numerical weight in the scheduler to favor larger jobs, so Alice's jobs are higher in priority and higher in prominence in the overall job list.

The overall job list is:

$$[A, B, C, D, E, F, G, H, J, K, M, N, P, Q, R, S, U, V, W, X]$$

. The ordering favors Alice's jobs because they are larger and that decision has been made by the administrators. There are no jobs scheduled for the first time slice (the system has just come out of a maintenance period, the system is empty, and there are no reservations), so the currently running/scheduled job list for this time slice is empty: $\{\}$.

5.0.1 First Time Slice: First Scheduling Pass. SFS tallies the total occupancy for each allocation based on the currently scheduled job list and then builds the first-pass local job list. The currently-scheduled list is empty, so all occuppancies are zero $OC(Alice) = 0$, $OC(Bob) = 0$. All current occupancies in the current time slice are less than target occupancies, so the local job list is the same as the overall eligible job list:

$$local[A, B, C, D, E, F, G, H, J, K, M, N, P, Q, R, S, U, V, W, X]$$

.

The first SFS pass determines job placement based on the local list. In the first job placement iteration, it places the top priority job, A, so the new local list is

$$local[B, C, D, E, F, G, H, J, K, M, N, P, Q, R, S, U, V, W, X]$$

and the running/scheduled list is $\{A\}$. Total occupancy (how we keep track if the machine is full for this time slice) is 200 (out of 1000 nodes)

Second job placement iteration: evaluate occupancies,

$$OC(Alice) = 200 < T^{target}_{Alice},\ OC(Bob) = 0 < T^{target}_{Bob}$$

, all occupancies are still below target, so the local job list doesn't change. Job B is highest on the local list, and it's placed in this time slice. After placement, the local eligible job list is

$$local[C, D, E, F, G, H, J, K, M, N, P, Q, R, S, U, V, W, X]$$

and the current running/scheduled list is $\{A, B\}$. Total occupancy is 400.

Third job placement iteration: evaluating occupancies, the Bob allocation is still below target, $OC(Bob) = 0 < T^{target}_{Bob}$, but the Alice allocation is now *above* target for this time slice:

$$OC(Alice) = 400 > T^{target}_{Alice}$$

so before further placement considerations, all jobs from the the Alice allocation are removed from the local eligible jobs list:

$$local[M, N, P, Q, R, S, U, V, W, X]$$

. After the occupancy considerations, scheduling continues. The next job placed is M; at the end of this iteration the local list is

$$local[N, P, Q, R, S, U, V, W, X]$$

and the scheduled/running list is $\{A, B, M\}$. Total occupancy is 450. 4th iteration: Occupancies:

$$OC(Alice) = 400 > T^{target}_{Alice}, OC(Bob) = 50 < T^{target}_{Bob}$$

. $local[N, P, Q, R, S, U, V, W, X]$, place top priority job N.

$$local[P, Q, R, S, U, V, W, X], \{A, B, M, N\}$$

. Total occupancy: 500.

5th iteration: Occupancies:

$$OC(Alice) = 400 > T^{target}_{Alice}$$

, now the Bob allocation has exceeded its target occupancy:

$$OC(Bob) = 50 > T^{target}_{Bob}$$

, so Bob's jobs are removed from the local list, which is now empty $local[]$. Since the local list is empty, there are no jobs to place, and the first SFS pass ends with scheduled/running list: $\{A, B, M, N\}$. Total occupancy remains 500.

5.0.2 First Time Slice: Second Scheduling Pass. The second scheduling pass continues where the first one left off, but now using the full job list unchanged by occupancy calculations. After the first scheduling pass, the eligible job list is

$$[C, D, E, F, G, H, J, K, P, Q, R, S, U, V, W, X]$$

and running list $\{A, B, M, N\}$. The second pass iterates through the job list, scheduling the jobs to run, until the machine is full or the list is empty.

Place job C,

$$[D, E, F, G, H, J, K, P, Q, R, S, U, V, W, X], \{A, B, M, N, C\}$$

, total occupancy 700.

Place job D,

$$[E, F, G, H, J, K, P, Q, R, S, U, V, W, X], \{A, B, M, N, C, D\}$$

, total occupancy 900.

The highest current priority job is E, but it's 200 nodes and there are only 100 nodes free.

$$[E, F, G, H, J, K, P, Q, R, S, U, V, W, X], \{A, B, M, N, C, D\}$$

. How the scheduler deals with this depends on the software. We'll assume that we have a reservation depth[5] of 2, so up to 2 jobs can be considered to be placed with the primary scheduling loop. Job E won't fit because it's 200 nodes and there are only 100 left. The next job in priority order is F, and can't be placed in the current time slice for the same reason. We've reached the bottom of the reservation depth, so the second scheduling pass ends.

5.0.3 First Time Slice: Backfill Pass. Now the scheduler does its backfill pass, to see if there are *any* more jobs that it can fill in, no matter their priority. The Alice allocation jobs won't fit in the remaining 100 nodes, but two of the Bob allocation jobs will. Jobs P and Q get scheduled, so all nodes are occupied for the first time slice, with an occupancy of 1000. At the end of scheduling for the first time slice, the running assignment list is $\{A, B, M, N, C, D, P, Q\}$ and the eligible job list is $[E, F, G, H, J, K, R, S, U, V, W, X]$.

6 EFFECTS OF SIMULTANEOUS FAIR-SHARE

This example illustrates several advantages of Simultaneous Fair-share over traditional numerical priority-based scheduling. First, SFS does not remove the authority of priority weighting to determine what jobs should go first. The priority weighting of this system is designed to prioritize jobs with high CPU counts, and that has been accomplished. In the final job configuration of both time slices in the example here, the 800 of the 1000 nodes in the example are occupied by jobs with the higher CPU count.

However, the Bob collaboration is still getting jobs through. It's accomplishing its work. This is why Simultaneous Fair-share is a powerful tool. It provides a reasonable fair-share behavior but unlike traditional history-based fair-share, it acts according to current information and thus it provides immediate behavior changes.

6.1 How SFS Damps the effects of Queue-stuffing

Earlier sections described the effects of queue-stuffing. Those effects will always be present when a system has a mix of groups who submit lots of jobs early, and those who need time do scale testing and debugging and then submit in a slower stream of jobs. Unlike linear priority scheduling, Simultaneous Fair-share gets both groups what they need. The group with the big initial submissions get served quickly. As soon as the slower group begin to submit jobs, their jobs get scheduled in the first scheduling pass up to their target throughput, so despite the rather full job queue, they see a very responsive system.

6.2 How SFS Addresses Under-served Job Classes

In addition to dealing with queue stuffing well, SFS also automatically addresses the problems of small linear job streams. As long as a group can run their streams within their target throughput, their jobs will be scheduled in the first scheduling pass, with good scheduling turn-around. When one job finishes, that leaves a hole in the system where jobs can be run, which in turn triggers a scheduling pass. Because of the simultaneous nature of SFS, that group's occupancy has at that point dropped by the size of the job that just finished. The first pass scheduling will need to place a job in that space, and that same group's next job will naturally fit perfectly. Sometimes there will be a job that will fit from another group, but fairly often, jobs will follow on directly from the first one. This is a natural mechanism that promotes linear job chains as a side consequence of the way SFS works and is another example of why SFS is a better algorithm than linear job scheduling.

7 WEAKNESSES OF SIMULTANEOUS FAIR-SHARE

SFS doesn't address all contingencies of absolutely all user submission strategies. The job scheduling algorithm can be exploited by users who deliberately structure their jobs to take advantage of pass one scheduling. For instance, users could schedule short-duration high-occupancy jobs to force the scheduler to drain the system, then their jobs would be eligible to run right after that. A traditional history-based fair-share might be a reasonable tool to deploy here. Any group that had too much total occupancy of jobs over the past several allocation period would have their jobs lowered in priority so that wasn't possible any more.

8 COMPARISONS TO OTHER SCHEDULING SCHEMES

Moab and other schedulers are obviously aware of the problems of queue stuffing and under-served job classes. They have some solutions to these imbalanced use cases (like history-based fair-share) but they don't solve the problems as fairly or as completely as Simultaneous Fair-Share. Moab's documentation, for instance, mentions methods to contend with queue-stuffing that could be harmful to throughput. The Moab documentation[3] suggests using idle job limits for this purpose: "The primary purpose of idle job limits is to ensure fairness among competing users by preventing queue stuffing and other similar abuses. Queue stuffing occurs when a single entity submits large numbers of jobs, perhaps thousands, all at once so they begin accruing queue time based priority and remain first to run despite subsequent submissions by other users.

Simultaneous Fair-Share shares some characteristics with LSF's "Dynamic User Priority"[4]; they're both ways of dynamically placing jobs according to the current state of running jobs. However, SFS is much simpler because it doesn't require dynamic re-computation of a user's priority when each job is placed. Also, the Dynamic User Priority concept is still embedded in the linear priority job ordering overall philosophy. The power of SFS is that it breaks out of that mold and allows a cleaner separation of job throughput from priority scheduling.

9 TESTING THE SIMULTANEOUS FAIR-SHARE ALGORITHM

Simultaneous Fair-share is a concept at the moment, not part of an operational scheduler. We have created a very simple schedule simulator using perl scripts to stand in for the components of a scheduler system. We then added synthetic users that imitated some of the production workflows that we have seen on the Blue Waters system over its history. We ran identical experiments with the same job inputs to test job throughput through the virtual system. The experiments were run four times, twice with a traditional linear priority scheduler with two different scheduling priority maps, and then twice again with the same priority maps but with a Simultaneous Fair-Share scheduler.

9.1 Scheduler Simulator

We created a scheduler simulation framework for the purpose of testing this algorithm. It uses flat files on disk to store job state information. The scheduler part doesn't actually manage processor resources, it just makes scheduling decisions, then other scripts propagate jobs from the eligible state to running state to finished state. The simulated machine runs in an accelerated clock, incremented by 5 "minutes" every time the main loop updates. In this way, several days of simulated system time can be simulated in tens of minutes. All of the jobs start when scheduled, and then run for for a random time between 70% and 95% of the requested system time. Each scenario runs several [hundred] times, and the results used to produce the analysis shown below.

This simulated scheduler has been configured to have the same general decision process and calculations as the real Moab scheduler on Blue Waters system. The jobs are not identical but similar to several of the common types of jobs that allocations run on that system system.

9.2 Machine and Job Simulation Setup

The relationship between the simulated jobs and the simulated machine are designed to mimic (with scaling factors) the relationship between the actual Blue Waters machine and common jobs that have been run on it. The simulated machine is 1400 nodes, which is roughly 16 times smaller than the 22,600 schedule-able nodes in the Blue Waters XE (CPU) section. The simulation runs over about two and a half weeks. For the first 7 days of the simulation, three different users feed jobs into the system. On day 1 through day 7, Alice feeds 12 250-node jobs into the queue. On days 1 through 7, Bob feeds 6 65-node jobs into the system. Finally, on day 7, Chris feeds a 750-node job into the system. The tests start on day 1 with an empty system, an empty job queue, and then immediately the first batch of Alice's and Bob's jobs are fed into the queue. The simulation propagates forward 5 minutes at a time.

9.3 Simulation Testing Results

These are the results for the four testing scenarioes.

9.3.1 Linear Scheduler, eligible time dominant. The first configuration is with a linear job scheduler with priority settings similar to Blue Waters. The eligible-in-queue weight is dominant over node count. When the simulation starts, Alice's jobs start right away because she has the largest jobs. Bob's jobs start about 1 1/2 days into the first week, and then the run on about a 1 1/2 or 2-day cadence. Bob's response isn't great, but it's not too bad. (However, if Bob's jobs were linear, they would half less than 50% duty factor so that would be a bad.) Chris's job gets submitted on day 7, but despite its very large size, because of the emphasis on eligible-time, Chris's job doesn't start for almost 7 days.

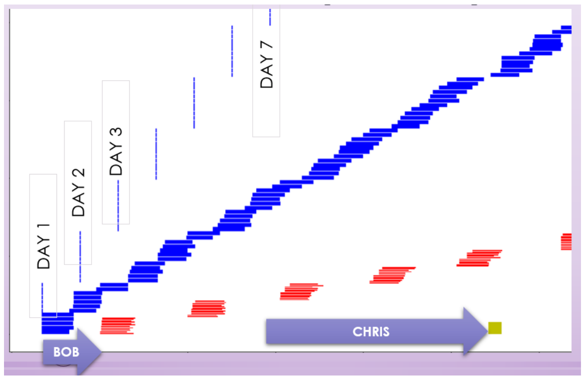

Figure 1: Scheduler simulation results. Linear priority scheduler, eligible-time dominant (close to Blue Waters stock scheduler configuration). Alice's jobs (blue blocks) start immediately, Bob's jobs (red blocks) start after about a day and a half latency, Chris's job (yellow block) starts with a 6-7 day latency.

9.3.2 Linear scheduler, node-count dominant. Node count is now dominant over eligible time. Alice's jobs start quickly as before. Bob's jobs are now pushed off more than 2 1/2 days, and his throughput of jobs is even lower. Chris's job does start fairly soon after its submitted due to the boost in node-count in the priority scheme. These last two sections are illustrative of the problem that linear priority scheduling carries with it. Depending on the job mix, the system administrators will always have to be adjusting the priority weights to make sure that everyone gets a good chance at getting resources on the system.

9.3.3 SFS scheduler, eligible time dominant. Original eligible-time dominant weights but with a Simultaneous Fair-share scheduler. Alice's jobs start running immediately. Due to SFS scheduling, Bob gets throughput immediately. This is great, especially if Bob's job topology is linear simulations that read each other's outputs. The SFS scheduler doesn't help Chris, here, though. Chris's job still waits multiple days to run. But this is largely because eligible-queue-time is really a solution for fair-share; it's designed to make sure jobs don't get stale. It's turned up too high. So for the last scenario, the balance of node-count will be increased compensate.

9.3.4 SFS scheduler, node-count dominant. For the last of our scheduling testing scenarios, we run the Simultaneous Fair-share scheduler, adjusted to favor large jobs over old ones. Alice's jobs, as always, begin immediately. As with the other SFS scenario, Bob's jobs being immediately and he has a steady throughput. The increase node-count priority, though, means that Chris's job now

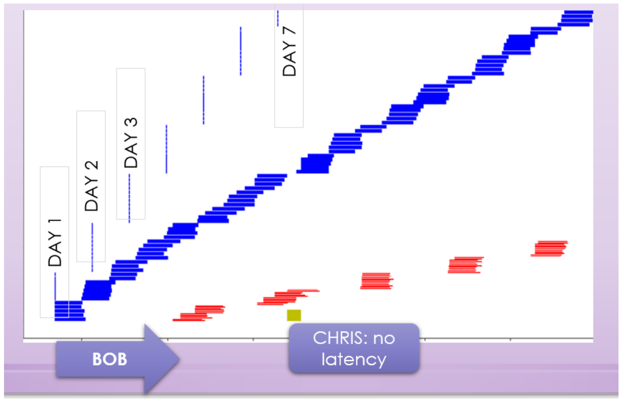

Figure 2: Scheduler simulation results. Linear priority scheduler, node-count dominant. Alice's jobs (blue blocks) start immediately, Bob's jobs (red blocks) start after about a 3-day latency, Chris's job (yellow block) starts in less than a day.

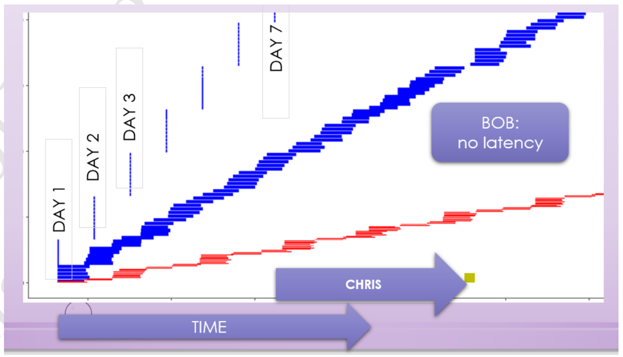

Figure 3: Scheduler simulation results. Simultaneous Fair-share scheduler, eligible-time dominant (close to Blue Waters stock scheduler configuration). Alice's jobs (blue blocks) start immediately, Bob's jobs (red blocks) start immediately, Chris's job (yellow block) starts with a 6-7 day latency.

starts within a day of when it was submitted. So Simultaneous fair-share schedule, with the priority scheme favoring high node-count jobs now favors high-node-count jobs (Chris and Alice) but also gives Bob's jobs good throughput.

10 NEXT STEPS FOR SFS RESEARCH

The very simple scheduler simulator was enough to show that the basic idea of Simultaneous Fair-share is viable and changed scheduling calculations as expected, but broader testing is required before deploying SFS on a real system. We are upgrading the simulation to use a database back-end rather than disk files so it can handle large job volumes. Once that's done, we will use weeks and perhaps months of (properly anonymized) job trace data from the Blue Waters system to test the SFS algorithm against the stock Moab scheduler handled the same job mix.

If SFS holds up to real scale and real job placement then we plan to create an SFS scheduling plugin for Slurm. Slurm has a

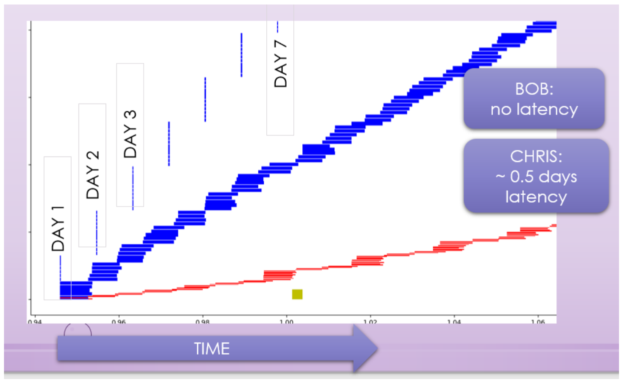

Figure 4: Scheduler simulation results. Simultaneous Fair-share scheduler, node-count dominant priority weighting. Alice's jobs (blue blocks) start immediately, Bob's jobs (red blocks) start immediately, Chris's job (yellow block) starts almost immediately.

plug-in architectures, so we will be able to create an SFS-driven job prioritization scheduler that will use the Slurm infrastructure. The documentation of the Slurm Scheduling plugin API[10] and the Slurm Resource Selection Plugin API[9], and statements elsewhere in the documentation[8] such as "By default, Slurm assigns job priority on a First In, First Out (FIFO) basis. FIFO scheduling should be configured when Slurm is controlled by an external scheduler." all suggest that a scheduler can be implemented as an external module using the Simultaneous Fair-share algorithm but using the Slurm infrastructure to handle the job launching. This would make it very easy to deploy SFS to any sites that use the Slurm scheduler, as the installation and infrastructure is already in place.

11 CONCLUSIONS

This paper has explained why linear priority scheduling has inherent weaknesses that made it very vulnerable to changes in HPC system job mixes. The new proposed Simultaneous Fair-Share algorithm would, if implemented, vastly improve the efficiency of scheduling HPC workloads and decrease the need for direct involvement of system administrators. While it's not a complete defense against deliberate manipulation of the scheduling algorithm by users, it would eliminate several classes of under-served jobs without having to specifically tune linear scheduling parameters for that task.

ACKNOWLEDGMENTS

This project is in the Blue Waters sustained-petascale computing project, supported by the National Science Foundation (awards OCI-0725070 and ACI-1238993) and the state of Illinois. Blue Waters is a joint effort of the University of Illinois at Urbana-Champaign and its National Center for Supercomputing Applications.

Thanks to the the Blue Waters Project Office and my immediate supervisor, Greg Bauer, who frequently allows me to charge off into the woods to slay dragons, even when the threat is not immediately apparent. Also thank you to the numerous members of the Blue Waters staff who have listened to my talks about this topic and offered questions, encouragement, suggestions, and especially criticism.

A ARTIFACT DESCRIPTION APPENDIX: A BETTER WAY OF SCHEDULING JOBS ON HPC SYSTEMS: *SIMULTANEOUS* FAIR-SHARE

A.1 Abstract

One of the options for including an artifact in a submission is a detailed description of a method or an algorithm. As such, since this paper is a detailed description of a scheduling algorithm, the paper itself could serve as the artifact. However, in case an additional functional code algorithm is required, I have included the source code tarball as well of the simulation as run. The version provided is an early proof-of-concept version that demonstrates the scheduling principles involved and is what generated the graphs shown in this paper. To run the simulation, follow the instructions in the README file in the top directory of the tarball.

A.2 Description

- **Program: scheduler simulation with driver scripts**
- **Execution: see the README file in the tarball**

A.2.1 How software can be obtained . This simulation is not released. Development tarball used for generation of the graphs in this paper provided with submission.

A.2.2 Hardware dependencies. No hardware dependencies.

A.2.3 Software dependencies. Only requires generic bash and perl, and python for the graphing.

A.2.4 Data-sets. This artifact consists entirely of a self-contained simulation made up of a collection of scripts. I doesn't have any internal data sets; all uncertainty is randomly generated using language random functions.

A.3 Installation

Untar the tarball. Follow the instructions in the README file.

A.4 Experiment workflow

Follow installation and run instructions in the README.

A.5 Evaluation and expected result

Use the sfs_histo_results.pl file on the output directory generated by the simulation to visualize the results. This won't quite produce the same display as the graphs in the figures but it will produce graphs with the same information that can be used to evaluate the simulation results.

A.6 Experiment customization

The README lists how to modify the simulation. You can modify the scheduling library file to set the scheduling priority weights, and modify

```
sfs_do_new_time_iteration.pl
```

to select which scheduling algorithm to use.

REFERENCES

[1] Adaptive Computing. [n. d.]. Moab Adaptive Computing Suite Administrator's Guide - v. 5.4. Retrieved August 23, 2019 from http://docs.adaptivecomputing.com/macs/6.3fairshare.php

[2] Adaptive Computing. [n. d.]. Using Moab Job Priorities âĂŞ Creating a Prioritization Strategy. Retrieved August 23, 2019 from https://www.adaptivecomputing.com/blog-hpc/using-moab-job-priorities-creating-prioritization-strategy/

[3] Adaptive Computing. 2014. Usage Limits/Throttling Policies. Retrieved August 23, 2019 from http://docs.adaptivecomputing.com/suite/8-0/basic/help.htm#topics/moabWorkloadManager/topics/fairness/6.2throttlingpolicies.html

[4] IBM. [n. d.]. IBM Platform LSF V9.1.3 documentation. Retrieved August 23, 2019 from https://www.ibm.com/support/knowledgecenter/en/SSETD4_9.1.3/lsf_admin/dynamic_user_priority_lsf.html

[5] Adaptive Computing Enterprises Inc. 2011. Moab Workload Manager Administrator Guide. http://docs.adaptivecomputing.com/mwm/archive/6-0/7.1.4reservationpolicies.php

[6] The Blue Waters Sustained-Petascale Computing Project. [n. d.]. Queue, Scheduling and Charging Policies. Retrieved August 23, 2019 from https://bluewaters.ncsa.illinois.edu/queues-and-scheduling-policies

[7] SchedMD. [n. d.]. Multifactor Priority Plugin. Retrieved August 23, 2019 from https://slurm.schedmd.com/priority_multifactor.html

[8] SchedMD. [n. d.]. Multifactor Priority Plugin. Retrieved September 27, 2019 from https://slurm.schedmd.com/priority_multifactor.html

[9] SchedMD. [n. d.]. Resource Selection Plugin Programmer Guide. Retrieved September 27, 2019 from https://slurm.schedmd.com/selectplugins.html

[10] SchedMD. [n. d.]. Slurm Scheduler Plugin API. Retrieved September 27, 2019 from https://slurm.schedmd.com/schedplugins.html

[11] SchedMD. [n. d.]. Slurm Workload Manager Version 19.05. Retrieved August 23, 2019 from https://slurm.schedmd.com/fair_tree.html