BLUE WATERS

SUSTAINED PETASCALE COMPUTING

November 22, 2019

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

HPCSYSPROS 2019 Workshop

Craig P Steffen csteffen@ncsa.illinois.edu

Blue Waters Science and Engineering Applications Support team

GREAT LAKES CONSORTIUM FOR PETASCALE COMPUTATION

Outline

- Linear Priority Job Scheduling
- Under-served Job Classes on Blue Waters
- Queue-stuffing (is NOT a User Problem)
- History-based Fair-Share
- Simultaneous Fair-Share (SFS)
- Scheduler Simulation: Testing SFS
- Shortcomings
- Future Work

terminology

- in all cases, users = “users and/or groups of users and/or allocations”

Linear Priority Job Scheduling

- Jobs are scheduled starting with the highest priority job and thereafter the next highest priority and so on
- The priority of a job is a linear sum of terms
- P(job) = A * x_a(job) + B*x_b(job) + C*x_c(job)
- A,B,C are weight coefficients set by the system administrators
- x_a(job) is that job's “a” characteristic at that time
- Typical characteristics are “size of job in nodes”, “total eligible time accumulated”, “requested duration”

Why Linear Priority Scheduling Falls Short

- Summing the effects of job characteristics together *couples* the effects of those characteristics to each other
- The weighing coefficients must be used to keep the effects in balance so that one job type won't dominate over another
- Shifts in submitter's aggressiveness and thus instantaneous job mix forces the admins to make changes to keep things flowing reasonably

Under-served Job Classes on Blue Waters

- Blue Waters has ~22,600 XE CPU nodes
- Jobs over 2000 get a reservation and run soon
- Jobs under 256 nodes backfill reasonably quickly and run
- Jobs between 256 and 2000 nodes, especially from 1000 to 1500, take a long time to gain enough priority to run (called “the valley”)
- Chained jobs allow other jobs to get scheduled between them
- Chained jobs in the valley are the worst served

Queue-Stuffing (is NOT a user problem)

- Some users pile large workloads all at once
- Some users have long strings of medium-sized sequential simulations
- Some users have intermittent groups of tiny short jobs
- Some users have to do R&D to get their workload set up, so have very few jobs but like short turnaround

All of these users behave reasonably. The scheduler should be able to treat their workload reasonably.

Linear weighted schedulers tend to favor one job flavor over another, so only one type of work tends to gets done. *This is not the user's fault;* ***it is a fundamental shortcoming of linear scheduling***.

History-based Fair-Share

- Schedulers have fair-share built in based recent history of finished jobs by a user or group
- Priority of user’s jobs are lowered when their recent history is above a threshold

Problems with History-Based Fair-Share

- History-based fair-share engages when completed jobs are registered, accounted for, and enter fairshare window
- Heavy use incurs drastic priority lowering, causing surging in use
  - heavy use for one cycle, no priority lowering, priority unchanged
  - a second cycle gets scheduled and launches before the first block has been registered in the fair-share algorithm
  - By the time the third cycle is being considered, that group's jobs have been heavily reduced in priority, as a result, no jobs by that group get scheduled
  - Lots of other jobs get to run in the vacuum
  - That group's jobs reurge in priority after the fair-share period, and they get scheduled again
  - the cycle repeats
- Even when it's not swamped by surging, fair-share is competing with the other priority weights for effectiveness, so it must be part of the give-and-take of adjusting priorities

What SHOULD a scheduler do?

- prioritize jobs according to priority weight policy
- deliver cycles to users who are ready to use them
  - (don’t let users be left out)

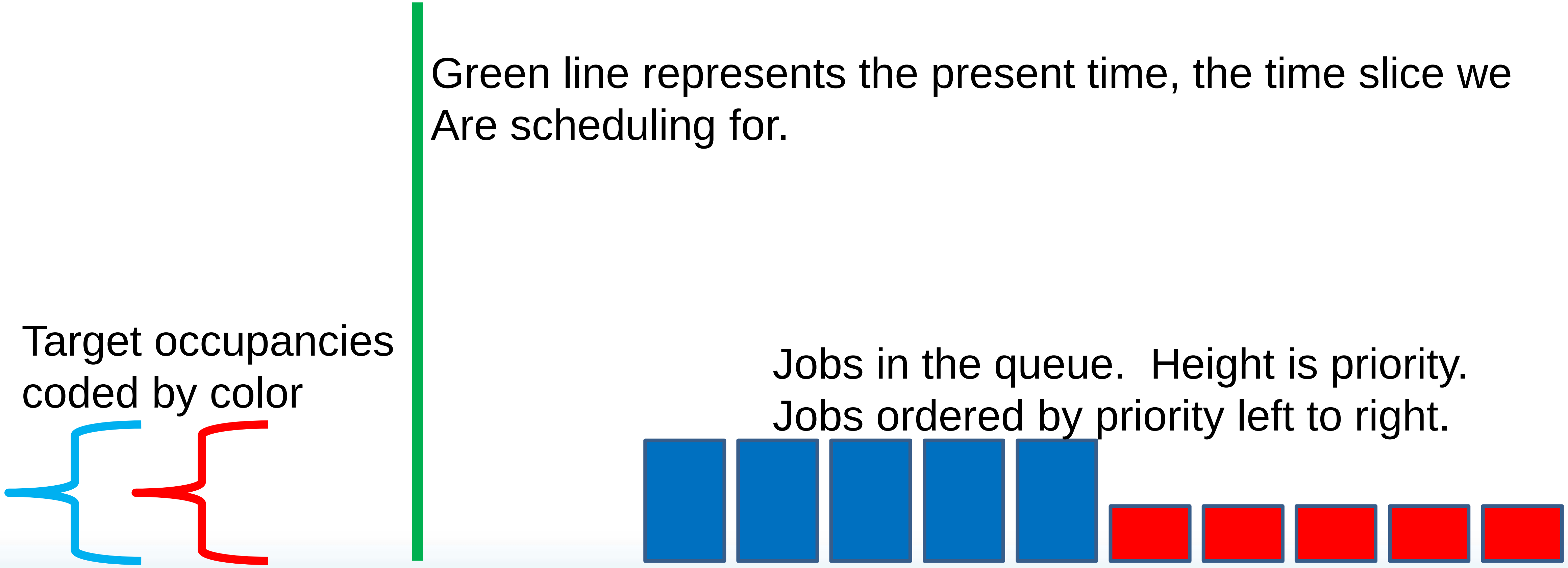
Simultaneous Fair-Share Algorithm setup

- The *Simultaneous* Fair-Share algorithm makes scheduling decisions based on the current, instantaneous resources a user or group is using (their *resource throughput*)
- User's *ideal* throughput: size of allocation / calendar time of allocation
- Example: Dr. Gajibe has 1,000,000 node hours for one year on Blue Waters Her ideal throughput is 1M nd*hr / 1 yr ≈ 114 nd*hr per hr or 114 nodes
- User's target throughput: ideal throughput * some factor (say, 2)
- Dr. Gajibe's target throughput (used in SFS calculations) is 114 x 2 = 228 nodes
- User's target throughput is STATIC in time; it does not change with time or as you use up your allocation

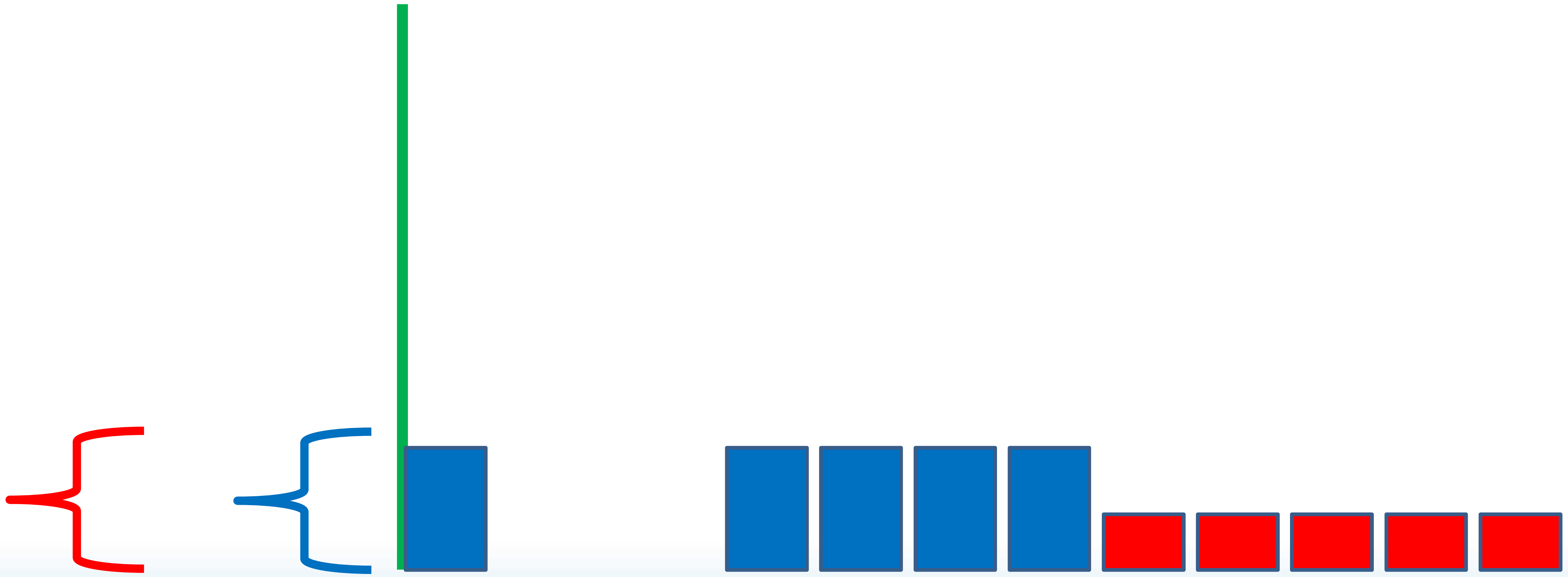
Simultaneous Fair-Share Algorithm procedure

- For any time slice there are TWO scheduling passes
- Scheduling pass ONE schedules for all users who *for that time slice* are not above their target throughput
  - occupancies are updated on the fly for that time slice. When scheduling a job for that time slice makes a user exceed their target throughput they are removed from the pass-one list job list
  - pass one ends when the pass one job list is empty or the machine is full
- Scheduling pass TWO uses the same job list but no consideration for target throughput

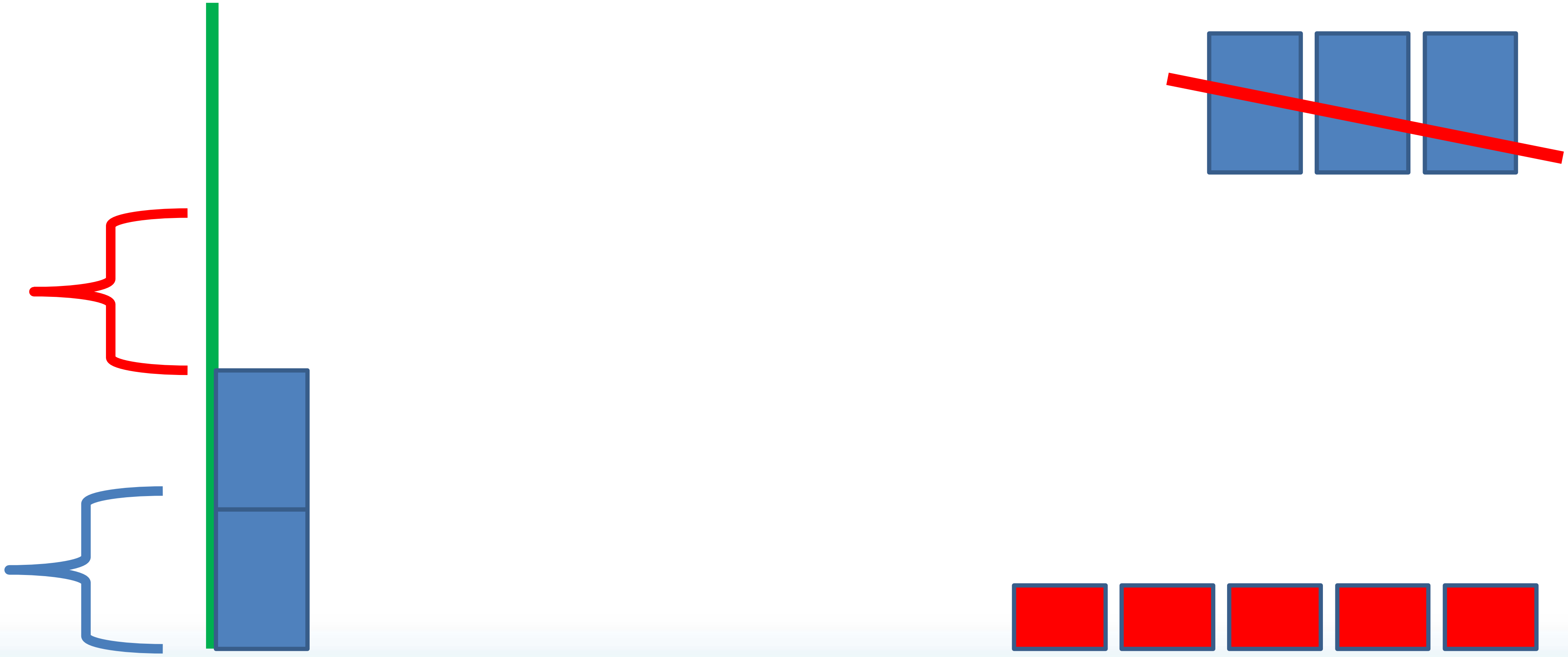
Simultaneous fairshare description: simple example: pass one. Jobs ordered within pass one by numerical priority value.


Green line represents the present time, the time slice we Are scheduling for.
Target occupancies coded by color
Jobs in the queue. Height is priority. Jobs ordered by priority left to right.

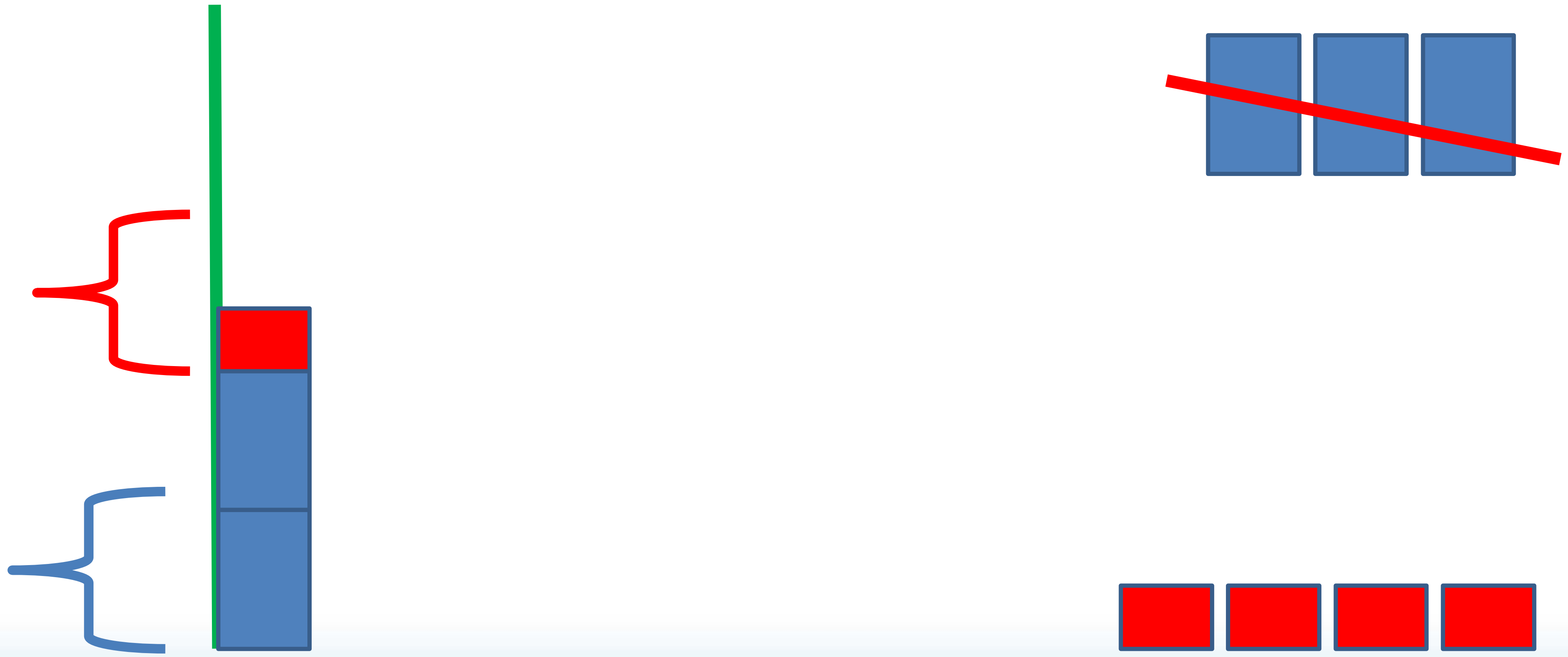
Simultaneous fairshare description: simple example; in pass one. One job scheduled. Red and blue still in pass one list.



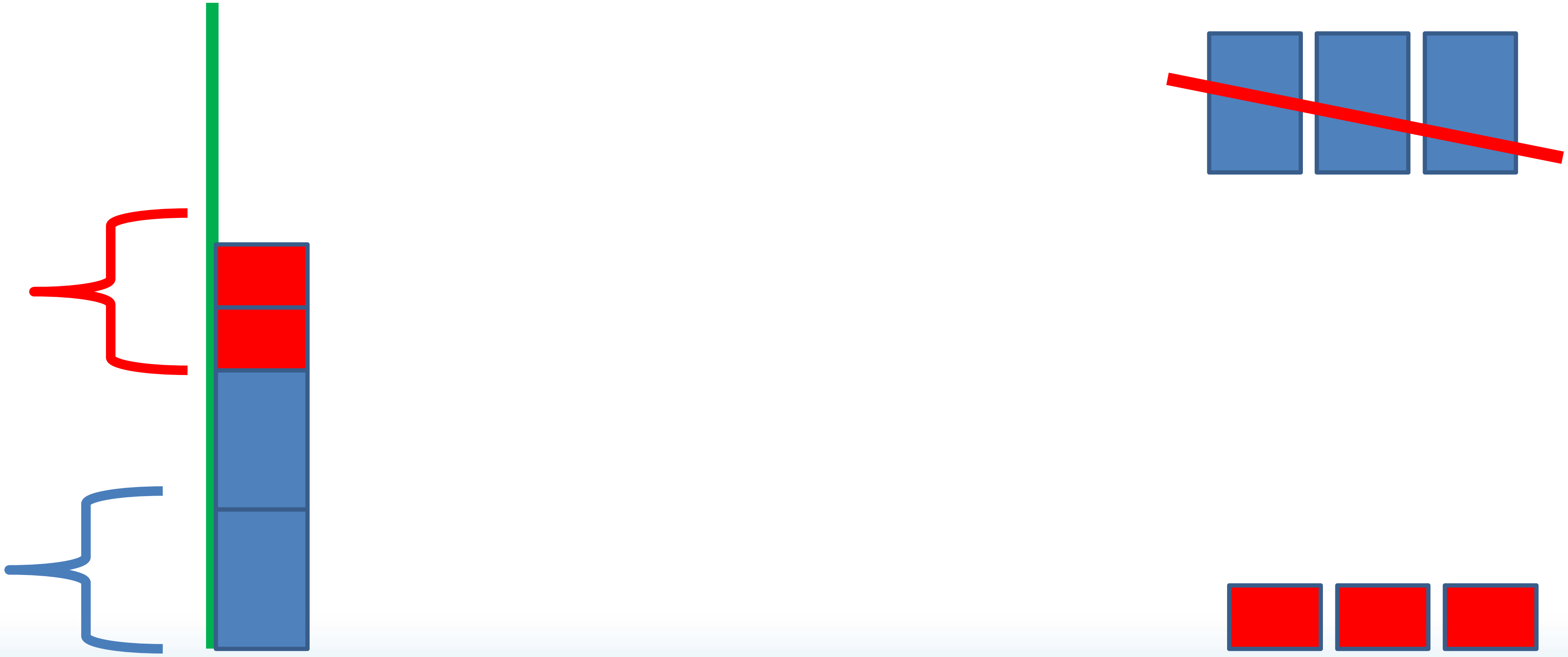
Simultaneous fairshare description: simple example; pass one, two jobs scheduled. Blue now exceeds target occupancy; blue now removed from pass one list.



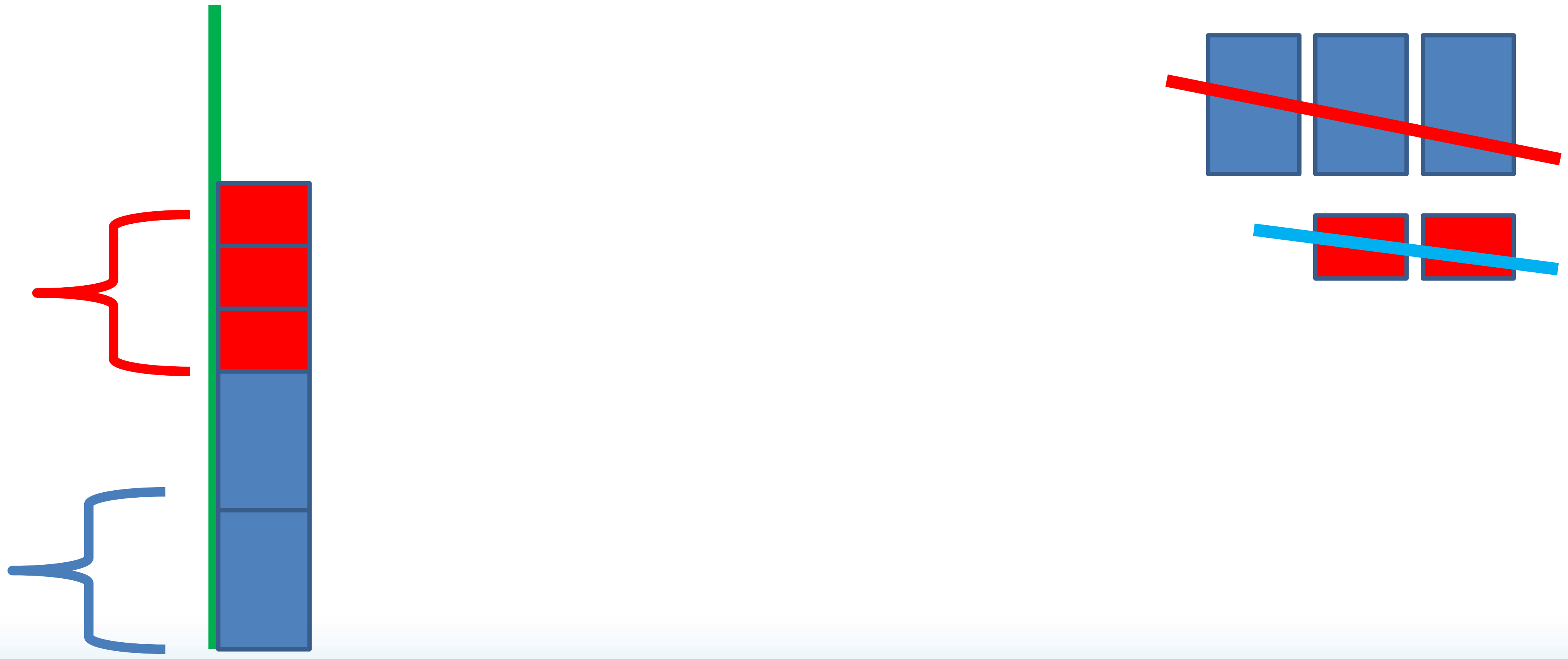
Simultaneous fairshare description: simple example. Pass one. Three jobs scheduled; red still eligible.



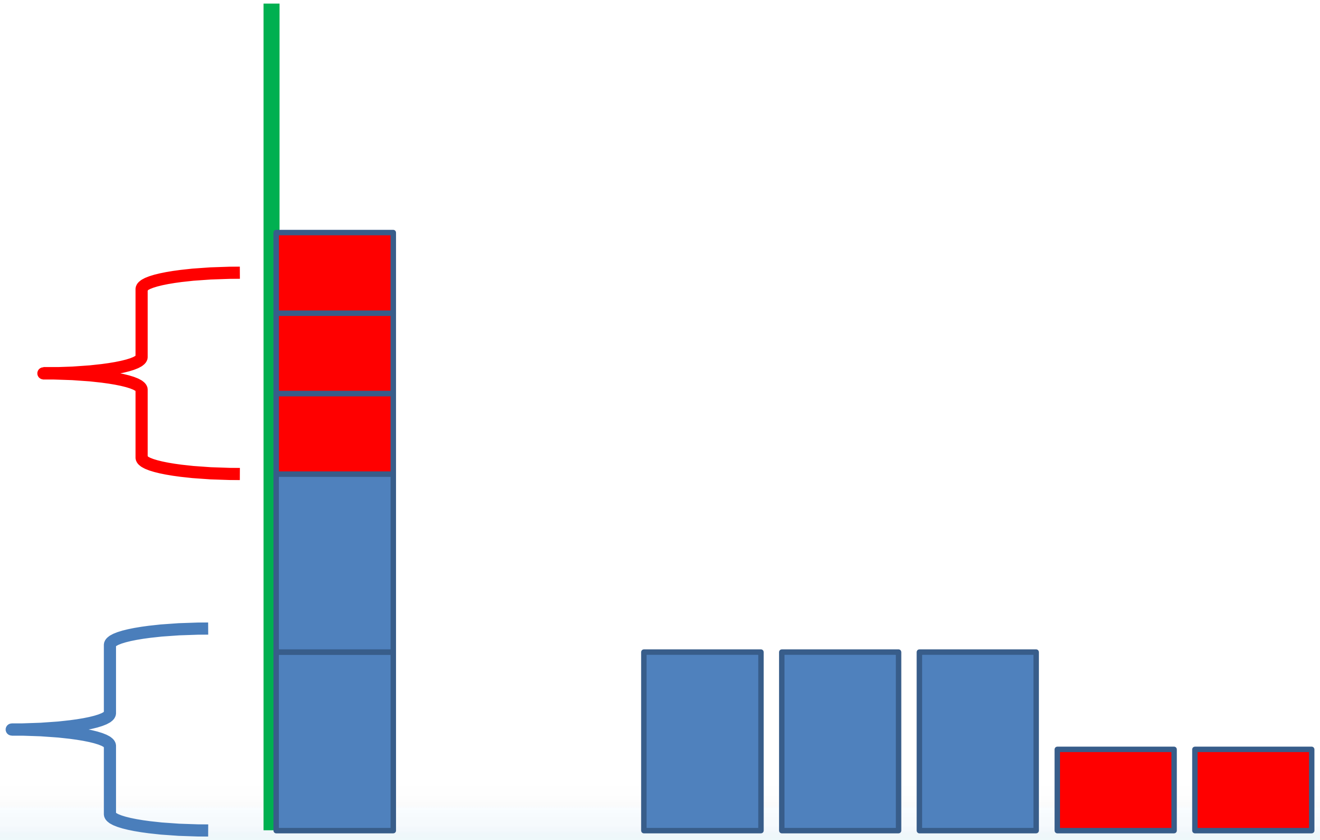
Simultaneous fairshare description: simple example, pass one. Four jobs scheduled, red still eligible.



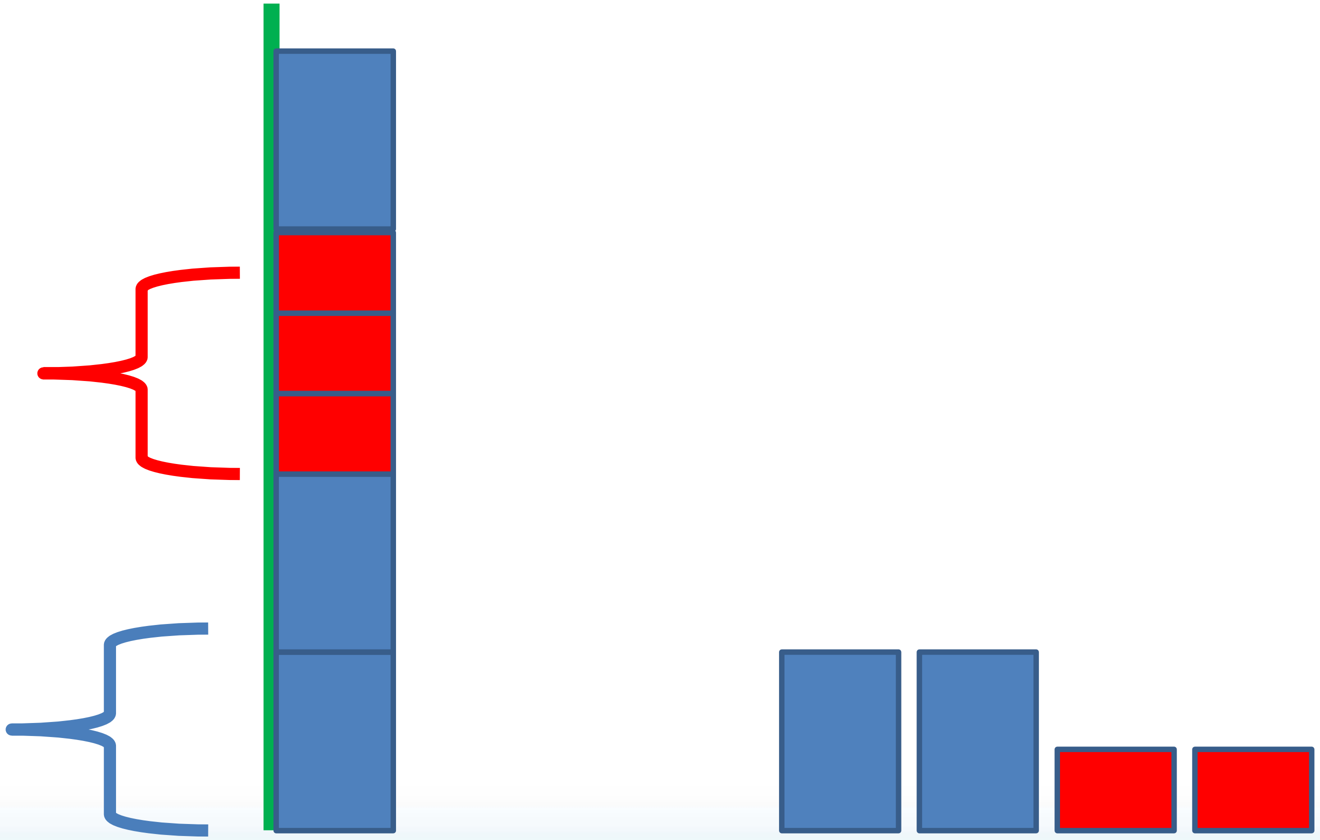
Simultaneous fairshare description: simple example. Red jobs now exceed red target; red jobs out of pass one list. Pass one list empty, pass one complete.



Simultaneous fairshare description: simple example: pass two. No eligibility removal. Jobs ordered by numerical priority



Simultaneous fairshare description:
simple example: pass two. One additional blue job scheduled. No more jobs fit. Pass two done



How Simultaneous Fair-Share Addresses The Problems with Linear Priority and History-Based Fair-Share

- SFS moves fair-share considerations to a different layer of scheduling than priority based on job characteristics
  - (It separates the who from the what)
- No fair-share oscillations because SFS applies separately to each instantaneous time slice

Testing SFS: Scheduler Simulation

- proof-of-concept, based on perl, writes state information to disk (ended up slower than expected)
- static set of timed user job submissions
- scheduler walks forward 5 minutes per iteration
- no topology-awareness

Testing SFS: Four Scheduler Setups

- 1) linear priority scheduling. Priority roughly equally weighted between node-count and time-eligible (approximate 2019 Blue Waters stock scheduler priority weights).
- 2) linear priority scheduling. Node-count weight significantly increased from Blue Waters stock.
- 3) Simultaneous Fairshare scheduling. Priorities Blue Waters stock.
- 4) Simultaneous Fairshare scheduling. Node-count weight increased from Blue Waters stock.

Testing SFS: Artificial Job Mix (applied to each config)

- Alice (blue jobs): on day 1,2,3,4,5,6,7: each day submits several 4000-node jobs
- Bob(red jobs): on day 1,2,3,4,5,6,7: each day submits several 500-node jobs
- Chris(yellow job): on day 7 submits one 14,000-node job

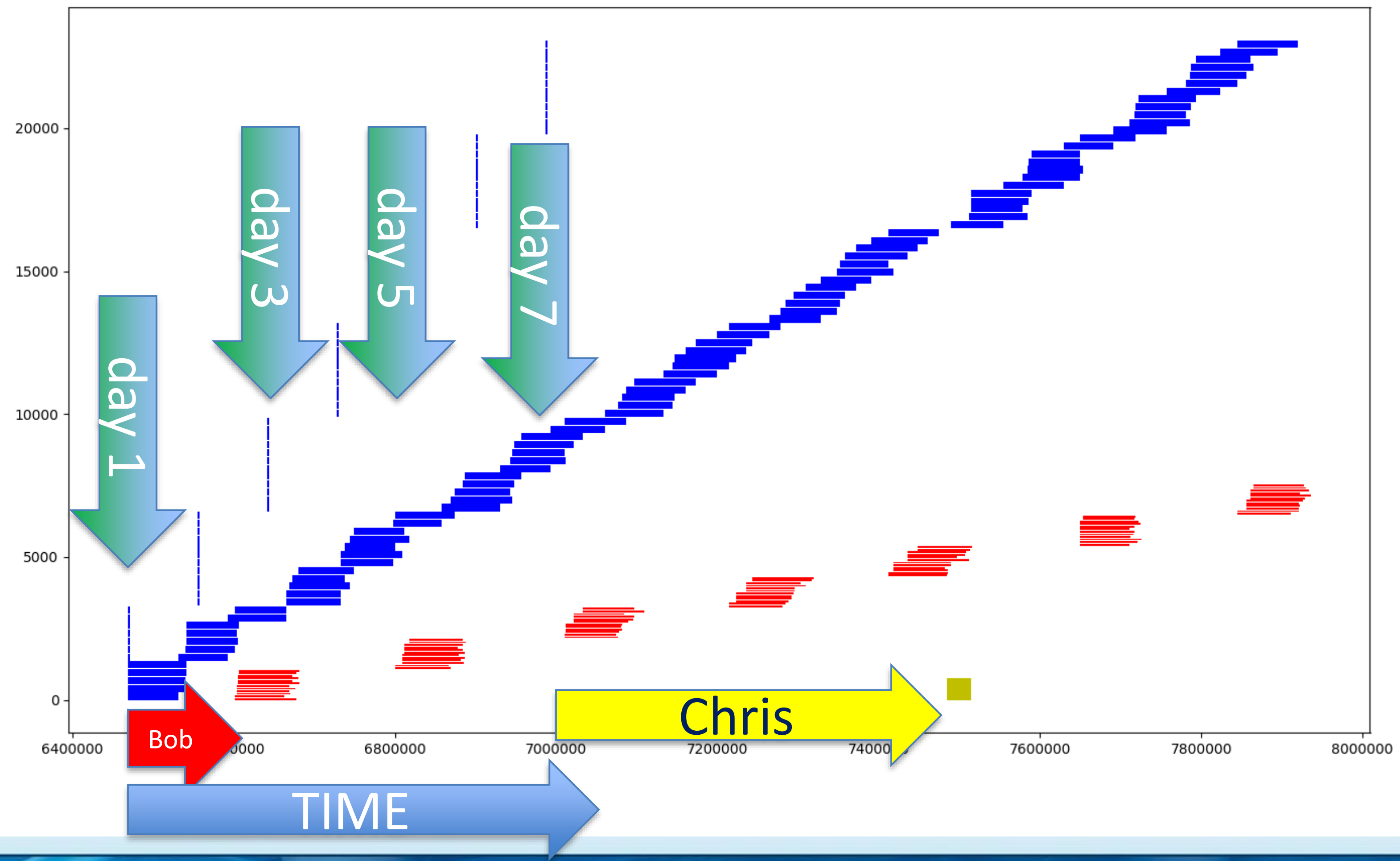
Simulation runs until the all jobs finish

Alice's jobs start immediately and have steady throughput

Look for:

- how soon Bob's and Chris's jobs start from when they're submitted
- how much throughput Bob's jobs get early on

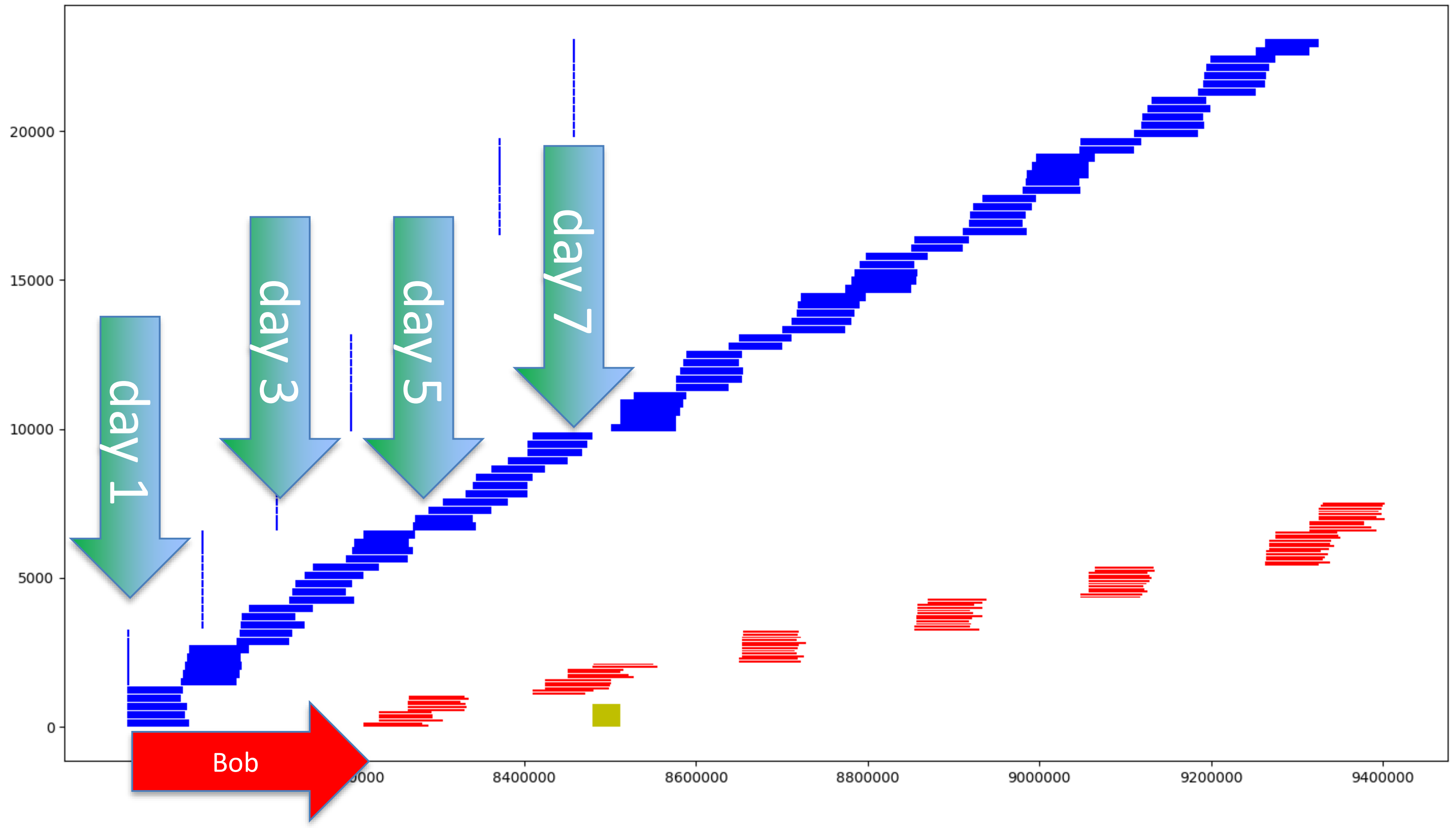
Testing Simultaneous Fair-share Results: Linear Priority Scheduling, Blue Waters stock priorities



Chris's latency ~7 days

Bob's latency: ~1.5 days no steady throughput

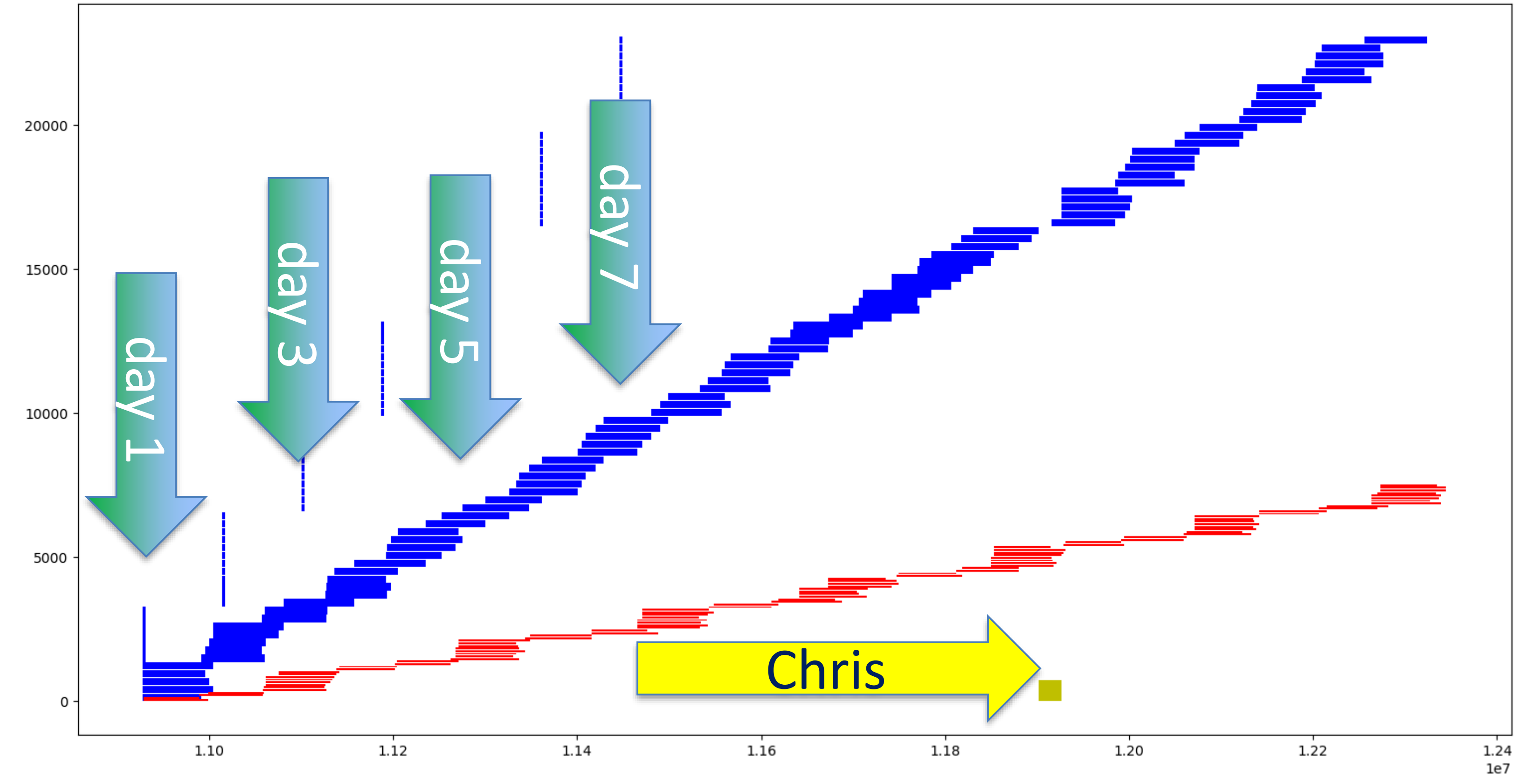
Testing Simultaneous Fair-share Results: Linear Priority Scheduling, Nodecount-Biased over stock



Chris's latency ~0 days

Bob's latency: ~3 days no steady throughput

Testing Simultaneous Fair-Share Results: SFS Scheduling, Blue Waters stock

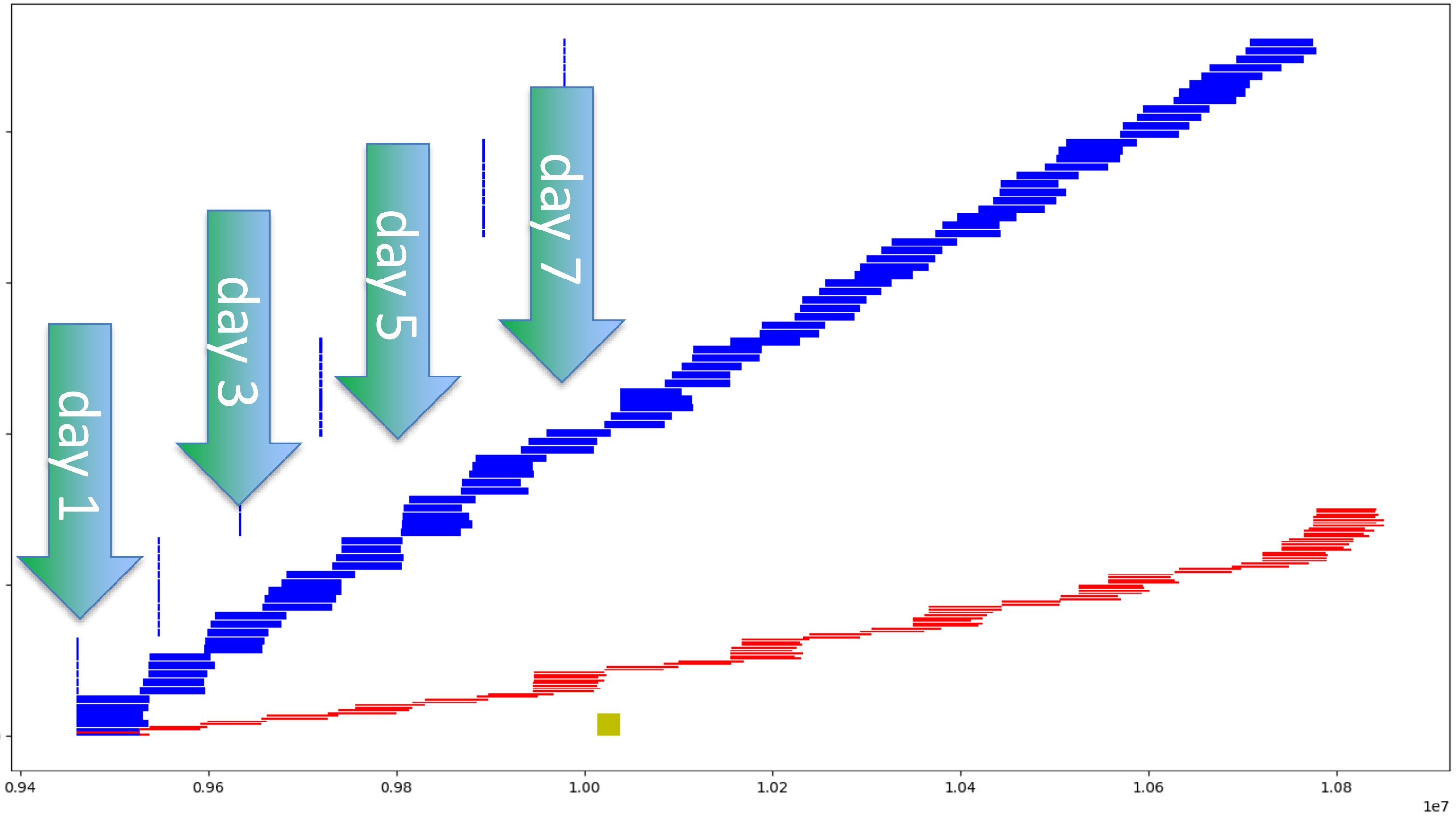


Chris's latency ~7 days

Bob's latency: ~0 days, steady throughput

BLUE WATERS
SUSTAINED PETASCALE COMPUTING

Testing Simultaneous Fair-Share Results: SFS Scheduling, Nodecount-Biased over stock



Chris's latency ~.25 days

Bob's latency: ~0 days steady throughput

What Simultaneous Fair-share Buys us

- Fair-share no longer has to be baked into job priority mix
  - naturally combats or eliminates underserved valley
- SFS naturally supports chained sequential jobs
- Users starting out (initial testing) have a natural mechanism to get jobs running quickly

How does SFS deal with TAS vs. backfill

- How to implement SFS as a full scheduler with topology awareness?
- Walk forward in time until there's space for top job *in that time slice's* pass one list
- (Testing this will require a more fully-featured scheduler simulator with topology-awareness)

What the Scheduler Simulator Does *Not* Do (yet)

- No topology awareness
- Not fast enough to cope with a real job throughput

What knobs do sysadmins have in SFS?

- Adjust overall ideal → target throughput multiplier
- If necessary to promote a user, adjust their individual ideal→target throughput multiplier

Weaknesses of Simultaneous Fair-Share

- Still subject to gaming by putting in ONE big job all the time
  - (possibly blunted by not having the incentive to do this?)
- Unfamiliarity

Future Work

- update the simulation to use databases to make it faster
- set up a month’s worth of real Blue Waters job submissions as a baseline test set
- use the test set multiple times to quantitatively investigate how different scheduler configurations (both linear and SFS schedulers) treat types of jobs
- Understand how to user SFS with topology-awareness
- Implement SFS as a Slurm plugin

Acknowledgements

- SFS work is part of the Blue Waters sustained-petascale computing project. Thank you to 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.
- Thank you to the Blue Waters Project Office and my boss, Greg Bauer for allowing me the time to work on this.