DOI 10.12694/scpe.v15i3.1020
Scalable Computing: Practice and Experience ISSN 1895-1767
Volume 15, Number 3, pp. 273–285. http://www.scpe.org

FAIRNESS OF TASK SCHEDULING IN HIGH PERFORMANCE COMPUTING ENVIRONMENTS

ART SEDIGHI[†‡] YUEFAN DENG[§] AND PENG ZHANG[¶]

Abstract. We claim that the current scheduling systems for high performance computing environments are unable to fairly distribute resources among the users, and as such, are unable to maximize the overall user satisfaction. We demonstrate that a user can game the system to cause a temporal starvation to the other users of the system, even though all users will eventually finish their job in the shared-computing environment. Undesired and unfair delays in the current fair-shared schedulers impede these schedulers from wide deployments to users with diverse usage profiles.

Key words: High performance computing, task scheduling, shared infrastructure, parallel computing, grid computing.

AMS subject classifications. 68M20

1. Introduction. Shared-computing environments where more than one user requests access to the underling system have to deal with many unknown aspects, such as how to deal with selfish behaviors of participants in real-time, which may result in low utilization of resources. There has been research in this area as it pertains to network routing [1], but very little has been done to consider selfish behavior in high performance computing (HPC) environments. With network routing, selfish behavior is more likely to backfire as many congestion control mechanisms simply drop packets and cause the originator to retry [2]. In shared-computing environments where task schedulers frequently use what is known as fair-share mechanisms [3] to schedule resources, users could game the system to monopolize resources and essentially "win the game" as in Game Theory. As such, game theory semantics describes our computing model nicely: users being the players of a game; strategies map directly with how a given task is submitted for completion and is scheduled; scheduler being the game mediator; and the amount of computing resources available to a user based on the current state of the infrastructure maps to the utility perceived by that user. A win is to gain a larger share of available resources. All users finish their jobs eventually, but the delays caused by competition are undesirable system events [4].

It is conceivable to treat a scheduler as the mediator to a zero-sum game. We assume the total number of resources available to a scheduler and as well as the number of users to be constant. As such, the number of resources that one user is assigned directly reduces the available resource pool for other users. Based on a user's task submission strategy, a user realizes a utility that is derived from the number of resources it is assigned by the scheduler. To address this classic problem, we adopt the concept of Nash equilibrium in game theory. Nash equilibrium represents the socially optimum solution, where no player is inclined to unilaterally change its strategy and increase its utility [5]. Strong Nash equilibrium further states that no subsets of players have the incentive either [6]. Based on this, if a user has the willingness to change his strategy and, as a result, change his perceived utility, we can deduce that we are not at an equilibrium point. Finding the solution to Nash equilibrium is not the goal of this paper but revealing the shortcomings of the fair share scheduler in shared environment is. Finding the Nash equilibrium point or the strong equilibrium point has been shown to be at least NP [7, 8, 9].

2. Background. HPC clusters are used for interactive workloads where immediate response for a workload is desired, and such workloads are broken into smaller tasks to be executed in parallel. A task is the smallest unit of instructions (atomic unit) that needs to be executed on a node. A job is an assembly of such tasks. A job cannot be marked complete until all tasks are complete. This is different than job-based systems, where a single job represents the entire workload and its completion means the completion of that workload. In a

[†]Corresponding author
[‡]Department of Industrial Engineering, Texas Tech University, Lubbock, TX 79409 (art.sedighi@ttu.edu)
[§]Department of Applied Mathematics and Statistics, State University of New York at Stony Brook, NY 11794-3600 and National Supercomputer Center in Jinan, Shandong 250101, China (yuefan.deng@stonybrook.edu)
[¶]Department of Biomedical Engineering, State University of New York at Stony Brook, NY 11794-8151 (peng.zhang@stonybrook.edu)

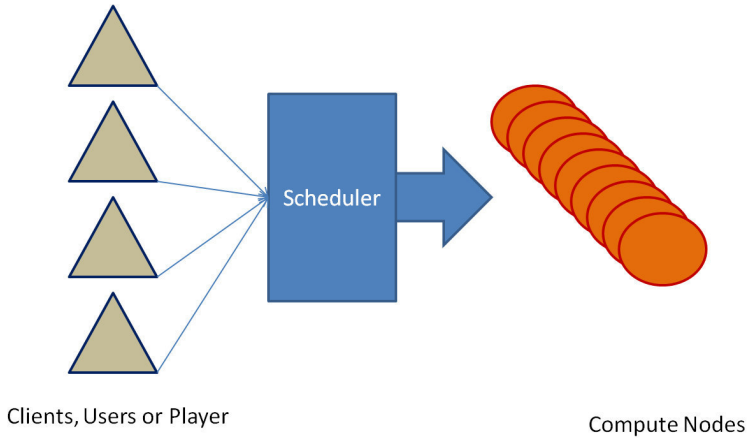

Fig. 2.1: The components of a shared-computing environment

task-based system, tasks are submitted over a period of time, and the method of submission has a significant effect on the overall completion time.

Interactive users continue to be attached to cluster until all tasks have been completed. Interactive workloads that run in the HPC environment are mostly single instruction multiple data (SIMD), where the same process (instruction) is executed on multiple data sets simultaneously. Depending on the size of the data set, and other factors [5], an imbalance could take place where not all the tasks complete at the same time. We will further show that this imbalance is further exacerbated by the scheduling mechanism.

HPC clusters achieve their desired high performance through the use of an integrated hardware, software and network [10, 11, 12, 13, 14]. A typical cluster contains 500 to 1000 physical nodes, but larger ones are possible [15, 16, 17, 18]. Considering a system with three participants in the aforementioned HPC cluster environment (Figure 2.1), we introduce the following terms:

1. Users: a client in need of computing resources such as a researcher in a biotech company, or a trader for a financial firm. The user interfaces with the HPC environment through a set of API's that facilitate communications with the environment. A rational user desires to utilize as many resources as possible without any regard to other users. Such greedy behavior may yield optimal solutions for one's own tasks, however, as many users share the same resource pool, creates temporary denial of access or starvation to other users.

2. Computing nodes (or resources): the servers that process the computing tasks. A node is the smallest processing unit and can be assigned to a single user. In game theory terminology, the computing nodes are the prize or outcome of the game. Each iteration of the game ends with the assignment of computing resources to each player, and as such the perceived utility of that user. The larger the portion of assigned computing nodes to a user, the higher the user's perceived utility. Since there are many tasks, and each task is short in duration, the utility of each user can change frequently during the game.

3. Scheduler (resource manager): the mediator that matches tasks with the available computing nodes optimal utilization of the HPC clusters.

The cardinality of a deployed infrastructure is ad-hoc in that there may be one or many of each of these components. HPC environments involve many users, most of whom are unaware of others' presence. Their decisions are based purely on selfish reasons. The goal of a given user is, and always should be, to game the scheduler in order to receive maximum resources to complete its tasks the quickest.

Scheduling systems used in HPC environments are based on queuing methodology where incoming queues are used to store tasks until they are scheduled for execution [19]. We are not introducing a new scheduling algorithm, but rather focusing on the inherent problem that is common to the queuing-based scheduling systems that use a fair share algorithm to distribute tasks.

Schedulers such as LSF [20], Sun Grid Engine (SGE) [21], PBS [22], Condor [23], Maui and Moab [24],

GridWay [25], Unicore [26] and GRMS [27] represent the de facto standard solutions that implement a First Come First Served (FCFS) policy for de-queuing tasks and scheduling for execution [19]. Even though the aforementioned systems implement the basic FCFS algorithm, these offerings also have the capability to prioritize based on the task characteristics [28, 29] such as Largest Job First, Smallest Job First (SJF), or Earliest Deadline First (EDF) [30], where a different priority is assigned to a given task according to the policy that the algorithm aims to maximize.

Most schedulers use a variation of the fair share algorithm [3] as the means of matching resource supply with user demand. For example, in a queuing system that uses the SJF policy, all the tasks of a given size (or runtime) is picked from the queue, and a fair share algorithm is applied to distribute these tasks across the available resources. In a queuing system that applies the basic FCFS queuing policy, the oldest tasks are taken off the queue and scheduled using a fair share algorithm. Fair share does so by dividing the resources through a ratio analysis of "the user who 'deserves' more, gets more". The amount that one deserves for a given time slot ($r_{c_k}(t)$) depends on the total number of tasks (T) pending to be completed.

$$r_{c_k}(t) = \frac{T_{c_k}}{\sum_{i=0}^{n} T_{c_i}} * R \quad (c_0 < c_k < c_n) \tag{2.1}$$

Subject to:

$$R = \sum_{i=0}^{n} r_{c_i}(t) \tag{2.2}$$

To simplify the ratio calculations without major drawbacks, we assumed that all tasks require the same computation, and all resources are homogeneous. Furthermore, we represent utilization in a binary format of "assigned" vs. "not unassigned". A given resource can go unassigned, i.e., idle, for a given scheduling cycle.

Let us consider the scheduler for two scheduling cycles (t_1 and t_2) where the scheduler takes a snapshot of the number of tasks of a given user waiting in a queue for a resource to be completed. Note that the resulting resource allocation depends only on the current state without knowledge of prior allocation. In order to drain the queues faster, the larger queue is assigned a larger portion of the available resources as demonstrated in Table 2.1. For example, for two users (c_1 and c_2) with c_1 having a queue size of 10 and c_2 a queue size of 90, user c_1 is assigned 10% of the resources while c_2 gets 90%. If $R = 6$, then we can demonstrate that c_1 is temporary starved.

Table 2.1: Fair-shared scheduling with temporal starvation of c_1

	t_1	t_2
c_1	$r_{c_1}(t_1) = \frac{10}{100} * 6 = 0.60 \sim 0$	$r_{c_1}(t_2) = \frac{10}{95} * 6 = 0.63 \sim 0$
c_2	$r_{c_2}(t_1) = \frac{90}{100} * 6 = 5.40 \sim 5$	$r_{c_2}(t_2) = \frac{85}{95} * 6 = 5.36 \sim 5$

As presented, the decision is not made based on any historical data. A snapshot of the current status is used to decide the next steps with the following assumptions:

- Running queue of pending tasks is an undesirable system event;
- The heavy users get more resources as they desire more;
- Temporal scheduling decisions are a good indication of the overall optimization problem.

The assumptions help drain the queue as fast as possible and, to achieve this, the heavy user is given an implicit higher priority. This static scheduling is considered fair-share in that each user will get its "fair share" of the system resources based on demand. Some scheduling policies use a policy-based mechanism, and solve the problem by introducing constraints in order to further aid the decision-making process.

We further assume:

1. There are many users, each with one or more jobs:

A typical environment has many users, each of which having its own collection of jobs that needs to be completed.

2. Each job is composed of many tasks:
In order to take advantage of the parallelism offered by the infrastructure, tasks must run in parallel, and when all tasks are completed, the overall job is completed. Essentially, tasks are atomic units to be scheduled by the scheduler. The rate of incoming tasks (e.g., 4 incoming tasks/sec) directly affects the pending tasks in the scheduling queue. A user has the ability to control the rate it submits tasks into the queue.

3. Users are unaware of others' behavior and strategies:
Although the scheduler decision system is static and fair, users are unaware of other users' behaviors. As such, each user will strive to get the most of what is available, and will act rationally and thus selfishly. We will further assume that tasks belonging to different jobs can compete with each other as well.

4. Limited resources where the total resources available is finite:
Usually, it takes much longer to acquire new hardware than the runtime of a given task. An argument is made about the ability to "spin-up" virtual environment, but virtual environments are still bound by the underlying constraints and availability of the hardware platform. Furthermore, the total number of resources is only a fraction of the total tasks pending to be executed. In our case, we assume we only have 500 resources, i.e., up to 500 tasks can run in parallel at any given time.

5. Selfishness and satisfaction:
Each user acts rationally in its desired completion of its job before others'. A user is acting rationally, if it is acting selfishly.

3. Rationale. An HPC cluster acts as a repeated zero-sum game with multiple users. The strategy that each user chooses in order to submit tasks directly affects the allocation of resources. In fair-share, the strategy chosen can starve some users and reduce the overall quality of the scheduler. In this type of environment one user's change in strategy directly affects the outcome of resource assignment. We show that shared environments with fair-shared scheduling algorithms never reach a Nash's equilibrium point [31] and that while some users are satisfied, maximum satisfaction is not perceived by all.

Our discussion revolves around a repeated game as with each new task, a new iteration of a game is played. We consider one task T_i to be a single iteration of a game denoted by J. The total time for a job t_J is then the total of the runtimes of all the tasks t_{T_i}.

$$J = \{T_1, T_2, T_3, \ldots, T_n\} \tag{3.1}$$

$$t_J = \sum_{i=0}^{n} t_{T_i} \tag{3.2}$$

More specifically:

$$\text{Runtime for a single iteration } (t_{T_i}) \ll \text{Total runtime for the entire game } (t_J) \tag{3.3}$$

As the total number of resources does not change, one user's submission strategy affects the number of resources that it is assigned. This in turn will affect all the other users. Nash equilibrium has been proven to be the social optimum point where no player can do better by unilaterally changing his strategy [5]; then if a user can do better by changing the strategy it submits tasks to the scheduler, the scheduler does not schedule in a fair manner. We will show that the scheduler can be gamed and "tricked" into starving a competing user, and that it is unable to cope with rapid changes in the submission profile.

4. Previous Work. Fair share algorithm was introduced in [3] and it was originally designed to allocate resources in time-sharing systems where the number of resources is limited and static. The research in this area [4, 6, 32, 33, 34] has been focused around job scheduling as opposed to task scheduling which represent a small part of an overall long-running job. Job fairness is measured based on actual start time as opposed to "could have" start times [35].

Paper [36] gives a good overview of the HPC schedulers available, with the majority of the schedulers under discussion being parallel task schedulers.

For the purposes of this research, we follow the scheduling taxonomy outlined in [37]. As such, we focused on the dynamic scheduling methodology [38], where jobs can and do come online dynamically. In such a

scenario, the goal of the scheduler then becomes maximizing resource utilization as opposed to lowering the overall runtime of the job [39]. We will show that this results in temporarily starving certain users, and as mentioned in the previous section, instills an imbalance.

Paper [40] argues that by increasing the number of resources in a given environment, the load imbalance would be reduced. Paper [41] uses priorities to alter scheduling decisions to reduce imbalance in the environment. We assume that all priorities are the same and do not change throughout the entire run.

Paper [4] considers restricted scheduling where a job must be assigned to a specific machine. We argue that a scheduler is optimal if it minimizes the social function: max total load, job latency, and weighted sum latency. In order to create a socially optimal solution, Nash equilibrium is considered as a basis of stability and the optimal social function. Paper [33] focuses on scheduling of jobs on identical machines. It also uses Nash equilibrium (NE) as the basis of fairness, but it assumes that jobs cannot be interlaced, whereas in our model, a machine could be executing task from various users over the course of time.

Paper [35] compares the scheduling methodologies and fairness of a supercomputing cluster versus a uniprocessor system. Since jobs are not further decomposable and cannot be preempted the cluster's fair-share algorithm cannot be fair, and not necessarily "is not" fair. Paper [32], in addition to considering batch job scheduling environment, uses the notion of social justice [42] as the measure of fairness. A schedule is fair in such a scenario if no job is affected by a later-arriving job. No research has been done for the topic of fairness as it pertains to task scheduling in shared environments with potentially conflicting interests.

Paper [25, 43] defines a congestion game to be a scenario where the payoff of a given player depends on the resource it is assigned, and how busy that resource may be based on how many other players are also on that resource. That paper, focusing on having identical machines and same-size jobs, argues that a scheduler may schedule a number of tasks on a given machine, if the number of pending tasks is greater than the total number of resources. As the number of tasks scheduled on a given machine increases – they are all competing for the same resource – the utility perceived by a given user could change based on how congested a resource becomes. This model does not work well with shared infrastructures and more specifically the model that each job is composed of a number of tasks (assumption 2). There could be other processes running on that machine, but the scheduler is strictly responsible for assigning one task to a resource.

In unrelated machines where the infrastructure is heterogeneous, [34] argues that congestion does not take place since the load of a given task is different on different machines. This model does not work in a shared infrastructure model as a job is completed if and only if all of its tasks have been completed. Based on this, unrelated machines cannot be treated any different than related machines. A shared resource infrastructure breaks down to its components of individual resources that need to work together to complete a set of tasks for a user. In short, there is an intrinsic commonality between related and unrelated resources in a shared environment. This commonality is the fact that each resource can contribute to the overall completion of a job.

There has also been research done around truthful algorithms as it pertains to scheduling tasks [44, 45]. In shared environments where there is little monetary incentive to be truthful about requirements, these algorithms do not work. In task-based systems, it is also very difficult to calculate the exact resource requirement as the tasks are each short in duration.

5. Methods. We modeled eight different cases, with each case representing a typical scenario on a system being managed by a fair-share scheduler. Each case compares two loads or submission strategies chosen by two competing users. We consider the following submission strategies in this paper:

- S_1: Normal-load submission strategy. This is the base load where all other strategies are compared against each other. For this strategy we assume that the user is submitting tasks at a constant rate throughout the simulation.
- S_2: Low-load submission strategy. For this strategy we assume that the user is submitting tasks at a constant rate but at some fraction of the normal strategy throughout the simulation.
- S_3: Sine-load submission strategy. For this strategy we assume that the user is submitting tasks in a start-stop fashion that resembles a sine wave
- S_4: Cosine-load submission strategy. For this strategy we assume that the user is submitting tasks in a start-stop fashion that resembles a cosine wave (time shifted sine wave)
- S_5: Burst-load submission strategy. For this strategy we assume that the user is submitting all tasks in a

very short amount of time and waits for results.

• S_6: Exponential-load submission strategy. For this strategy we assume that the user is submitting tasks at an increasing rate over time.

• S_7: Early-riser-load submission strategy. For this strategy we assume that the user is submitting all tasks in a very short amount of time before other users enter the system.

• S_8: Delayed-load submission strategy. For this strategy we assume that the user starts submitting tasks after all other users have entered the system.

The most basic load is denoted as the normal load submitted by c_1 where 1000 tasks are submitted by the user per time interval t, and that load does not change during simulation. Sine, cosine and step function workloads are aimed at demonstrating a start-stop workload where the user submits a certain number of tasks; waits and then submits another batch of tasks. Such loads are more indicative of real-world applications as a user may need to access an external data source (like a database for example) to gather the task data to submit. The exponential load will demonstrate the long-term effect of a shared infrastructure.

Case 1: Burst Workload where one user's workload is steady and low, and the other user comes in bursts. For purposes of keeping track, we have designated c_1 and c_5 to represent these two users.

Table 5.1: Task Submission Strategy for Test Case 1

Users	**Load Strategy**	**Load in** $\frac{\textbf{tasks}}{\textbf{time slice } (t)}$
c_1	S_1	$L_1(x,t) = 1000 * L_0(x,t)$
c_5	S_5	$L_5(x,t) = 10 * L_1(x,t) * U(t-a+b)$ $U(t-a+b) = \begin{cases} 1 & a < t < b \\ 0 & \text{otherwise} \end{cases}$

Where:

$$L_0(x,t) = 1$$

Case 2: Complementary workload where the timing of two users' workload complement each other. In this scenario, one user is idle while the other user is submitting tasks. For purposes of keeping track, we have designated c_3 and c_4 to represent these two users.

Table 5.2: Task Submission Strategy for Test Case 2

Users	**Load Strategy**	**Load in** $\frac{\textbf{tasks}}{\textbf{time slice } (t)}$
c_3	S_3	$L_3(x,t) = \max\{L_1(x,t)\sin(\pi t/2), 0\}$
c_4	S_4	$L_4(x,t) = \max\{L_1(x,t)\cos(\pi t/2), 0\}$

This type of workload is best suited for a shared environment as one user's busy period complements another user's idle time.

Case 2a: Complementary workloads with unbalanced profiles where the timing of two users' workload complement each other, but one user's workload is higher than the other's.

Case 3: Conflicting workloads where one user is steady but the second user is increasing its workload over time.

This case could conceivably be applied to a HPC environment where with the passage of time, more users join the environment with c_3 representing a single user and c_7 representing the load from the other users that

Table 5.3: Task Submission Strategy for Test Case 2a

Users	**Load Strategy**	**Load in** $\frac{\textbf{tasks}}{\textbf{time slice } (t)}$
c_3	S_3	$L_3(x,t) = \max\{L_1(x,t)\sin(\pi t/2), 0\}$
c_4	S_4	$L_4(x,t) = \max\{2 * L_1(x,t)\cos(\pi t/2), 0\}$

Table 5.4: Task Submission Strategy for Test Case 3

Users	**Load Strategy**	**Load in** $\frac{\textbf{tasks}}{\textbf{time slice } (t)}$
c_2	S_2	$L_2(x,t) = \frac{L_1(x,t)}{10}$
c_6	S_6	$L_6(x,0) = 1000 * L_0(x,t)$ $L_6(x,t) = L_6(x,t-1)^{1.01} \quad t \geq 1$

continually onboard the environment. The exponential load can be thought of as a way of demonstrating the rest of the users yet to come.

Case 4: Steady workloads where the two users have a steady stream of workloads that enter the system.

Table 5.5: Task Submission Strategy for Test Case 4

Users	**Load Strategy**	**Load in** $\frac{\textbf{tasks}}{\textbf{time slice } (t)}$
c_1	S_1	$L_1(x,t) = 1000$
c_2	S_2	$L_2(x,t) = \frac{L_1(x,t)}{10}$

Case 5: Early-riser workloads where one user enters the system early in order to claim resources.

Table 5.6: Task Submission Strategy for Test Case 5

Users	**Load Strategy**	**Load in** $\frac{\textbf{tasks}}{\textbf{time slice } (t)}$
c_7	S_7	$L_7(x,t) = 10 * L_1(x,t) * U(t-a)$ $U(t-a) = \begin{cases} 0 & a \leq t \\ 1 & 0 \leq t \leq a \end{cases}$
c_8	S_8	$L_8(x,t) = L_1(x,t) * U(t-b)$ $U(t-b) = \begin{cases} 1 & b \leq t,\ a \leq b \\ 0 & \text{otherwise} \end{cases}$

The total number of pending tasks waiting to be scheduled $Q_i(x,t)$ depends on the number of tasks that were scheduled and completed in the previous time slot.

$$Q_{c_i}(x,t) = L_{c_i}(x,t-1) + L_{c_i}(x,t) - r_{c_i}(t-1) \tag{5.1}$$

We further assume in our model that a given task T_k can be completed during the given time slot t, and that all the tasks are completed successfully. Introducing failure in our model would simply increase $Q_i(x,t)$,

and we are modeling that through known loads presented in this section.

6. Results. We will demonstrate the short-term effects of choosing a different submission strategy, followed by the long term effects. We will further elaborate on the scenario with limited number of tasks versus infinite tasks submitted by each user. Our simulations, unless otherwise expressed, ran under the following conditions:

Assume that $J(c_i) = \{T_1(c_i), T_2(c_i), \ldots, T_n(c_i)\}$ is a job set of user c_i. Function $T_j(c_i)$ means a job of user c_i with execution time T_j. $\|J(c_i)\|$ is the number of elements in the set J, that is, the number of tasks of user c_i. Based on these, we can see:

- Maximum number of tasks for both users was the same and capped to $T_{\max} = 100{,}000$

$$\|J(c_i)\| \leq T_{\max} \quad \forall i$$

It means for any user c_i, the number of tasks allowed to submit is no more than $T_{\max}$.

- All the tasks are the same length, and can be executed in a time slot

$$\forall_T \quad t_{T_i} = a$$
$$T_j(c_i) \equiv T_J \leq T_a \quad \forall j$$

$T_j(c_i)$ is referred to as the execution time of job T_j of user c_i and T_J is a constant, that is, the execution time of each and every job in the set $J(c_i)$. Thus, it means all the jobs in set $J(c_i)$ have a constant execution time, i.e., all the tasks in the set are of the same length. T_a is the length of a single time slot. Therefore, each and every task can be executed within one time slot.

- Simulation runs for 100 time slots

$$t = \{1 \ldots 100\}$$

Assume that $T_a(n)$ means the n-th time slot and thus here $n \in [1, 100]$.

- Scheduling takes place at the beginning of each time slot and its duration is negligible
- Total number of available nodes is constant throughout the simulation and is set to 500.

Case 1: For the first simulation, we modeled a scenario where one user is submitting tasks at a steady rate and at a time later, a second user submits the same number of tasks but in a much shorter time frame. Figure 6.1 depicts the input task profile for the two users. As can be expected, all the available nodes are dedicated to User 1 to start. As soon as User 5 starts submitting, the resources start shifting to User 5 (Figure 6.2), and it stays that way until the pending task size of the two users start to match up (Figure 6.3).

Case 2: This simulation models the case where the two users have complementary workloads. c_3 and c_4 are the two users tested, and one user's task submission takes places after the other user has completed submitting its tasks to be executed. The number of tasks submitted by the two users is exactly the same for each time slot for the initial case (Figure 6.4), and it clearly demonstrates how the sharing of resources can take place with pure complementary workloads. The resource distribution stays consistent throughout the simulation (Figure 6.5).

Case 2a: We extended this simulation with one of the users (c_4) having 2× the tasks per time slot. At the beginning of the simulation, as expected, the resource allocation was fairly dynamic in the way resources were shifted from one user to the next. As time went on, and as the pending task queue trumpeted the incoming task rate, the resource allocation became more static in that c_4 became the more dominant (Figure 6.6) receiver of resources.

Even with complementary task profiles, if rate of incoming task rate is not equal, the pending task queue inequality will cause one user to dominate the resource pool.

Case 3: This case deals with an environment where one user's workload is a very small fraction of the overall workload submitted by other users, (Figure 6.7). This is true as more users are on-boarded to the environment. Case 3 deals with this scenario, and demonstrates the resource allocation for c_2 as the number of tasks increase actually begins to decline (Figure 6.8).

Case 4 is a trivial case where the two users are sending tasks at a steady rate, but one user's task submission is lower consistently. The resource allocation does not change throughout as expected (Figure 6.9).

Case 5: this scenario represents a user entering the system earlier than the other user. Both users are submitting the same number of tasks to the system, but c_7 does so in a much shorter timeframe (1/10 of the

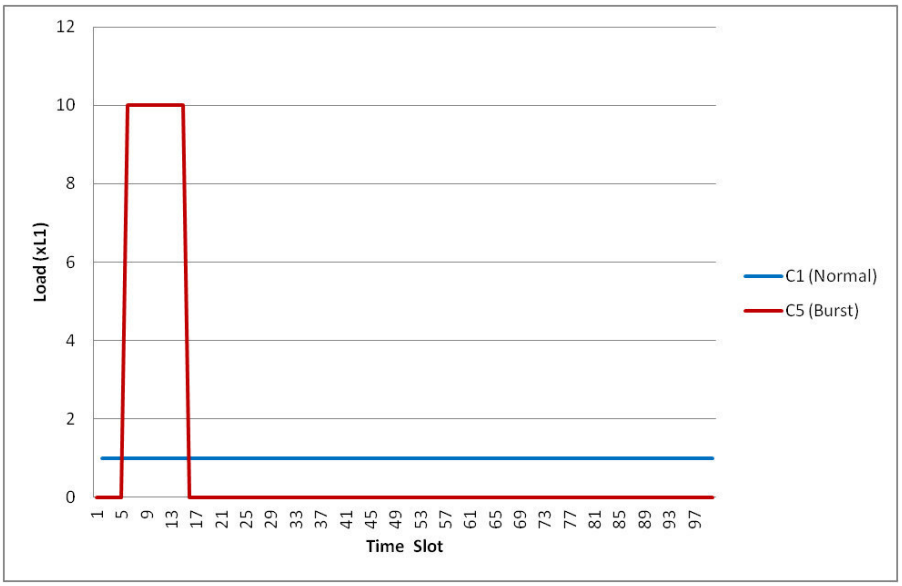

Fig. 6.1: Input task profile for Case 1 (Load is the multiplier of L_1)

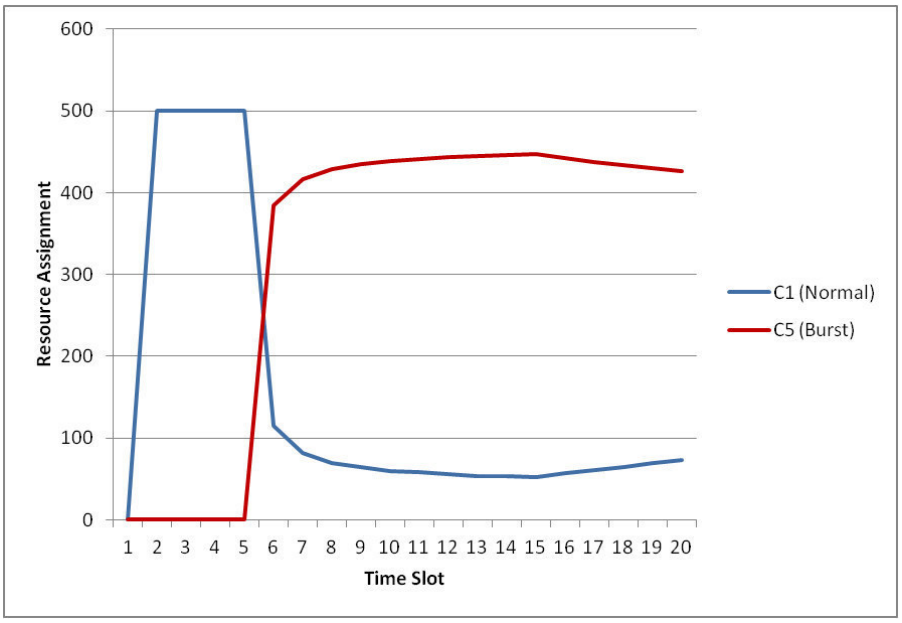

Fig. 6.2: Resources shift to User 5 and starve User 1 ($t = 1 \ldots 20$)

total submission time) (Figure 6.10). The resources are dedicated to c_7 until c_8 builds up a queue of equivalent size (Figure 6.11).

7. Analysis. Fair share does well with like workloads. As such, it intrinsically normalizes the workloads to make them alike. In doing so, the pending task queue of users with few pending tasks tend to suffer. This situation creates a false-sense of fairness and allows power players to monopolize the system.

Case 1 showed a typical simulation where steady workload is derailed by a user capable of overloading the

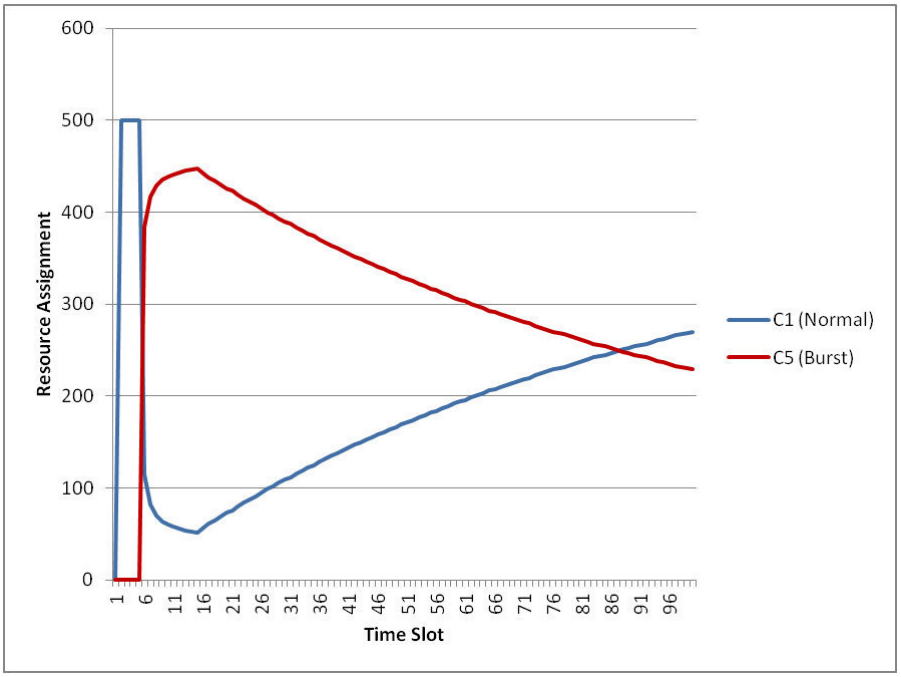

Fig. 6.3: Resource Allocation for Case 1

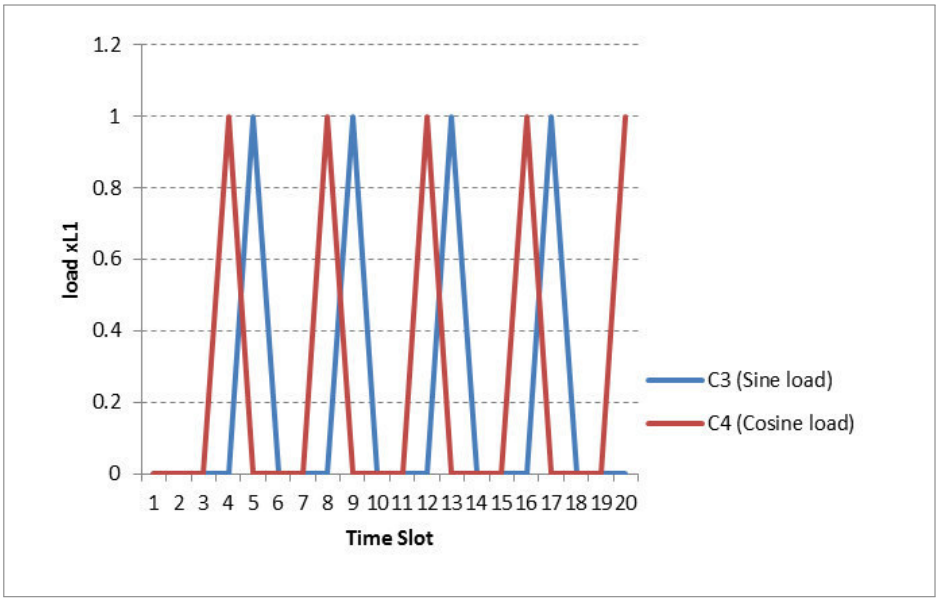

Fig. 6.4: Input task profile for Case 2 (Load is the multiplier of L_1)

system. User c_5 has the ability to burst tasks into the system in a short amount of time, and is thus capable of gaming the scheduler to assign the majority of the resources to it until its queue is drained. Case 2 shows complementary work profiles that qualify for a perfect sharing of resources. On the other hand, if the job profiles are not exactly the same as in case 2a, the long-term effect will be one of the users being starved.

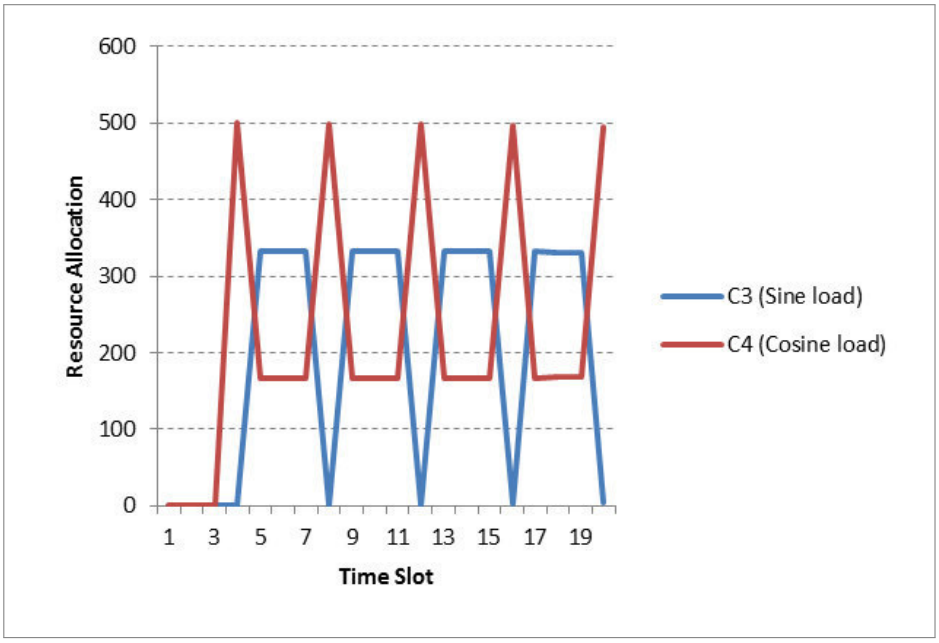

Fig. 6.5: Resource Allocation for Case 2

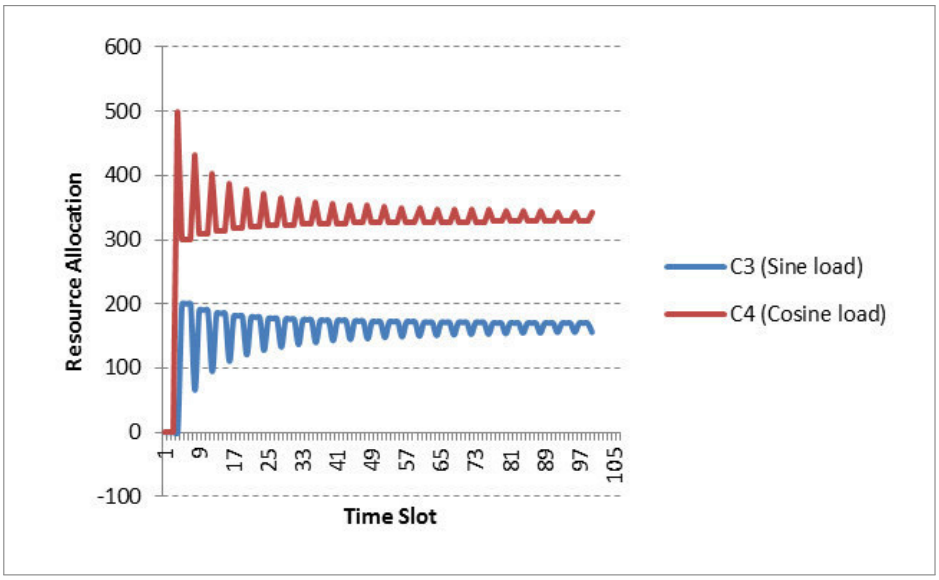

Fig. 6.6: Resource Allocation for Case 2a

Case 3 could represent a new user entering an already resource-lacking system. When there are many users waiting in the queue to be processed, a new user has a very tough time getting new resources. In heavily shared environments, cases 2a and 3 could become devastating to a new incoming user. Case 4 is where a fair-share scheduler is actually fair; steady workloads that span long periods of time.

Case 5 shows a late comer to a system that is already experiencing large queues. It shows that until the queues are drained, a newcomer will not get adequate resources.

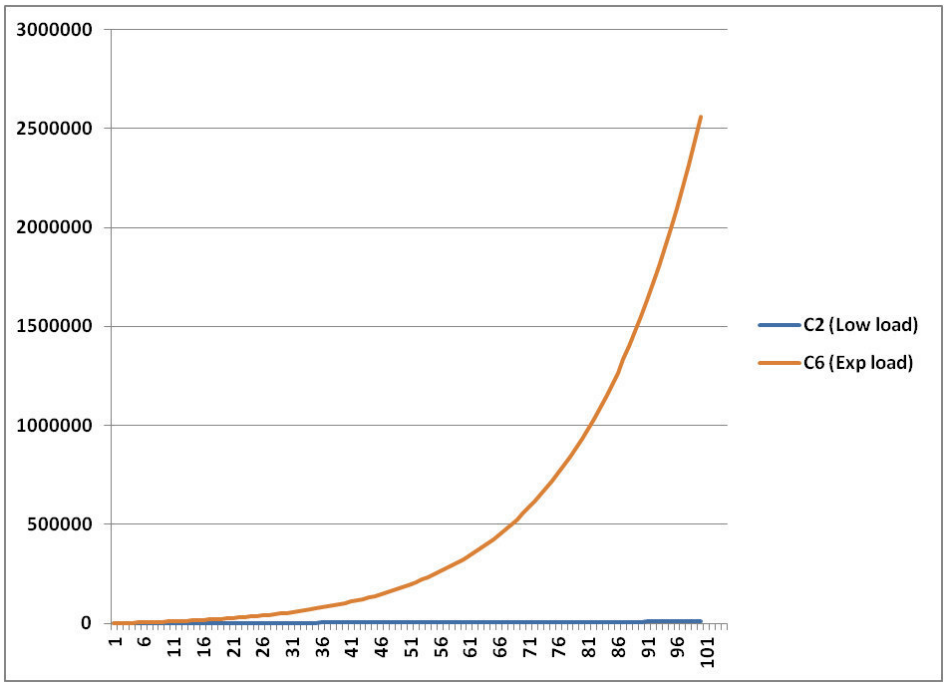

Fig. 6.7: Workload Comparison for Case 3

Fair-share algorithms are used widely as the most efficient way of scheduling tasks in HPC environments. We demonstrated in this paper that fair-share algorithms fail to do so in cases where the workloads vary in any significant way.

8. Conclusions and Future work. We revealed the shortcomings of a typical fair-share scheduling system for HPC environments. We demonstrated that fair-share schedulers normalize the queue depth first before being fair to all users. A fair-share scheduler requires that all the workloads be the same, all have the same start-time, all have the same end time, and all submit tasks to the system at the same rate. Any deviation from this will result in the scheduler having to compensate for the inequality by assigning more resources to the user[s] in a manner to create normalized workload across all users. This approach works for steady workloads where the request profile does not change. In such a scenario, fair-share schedulers show fair resource division across the users. For a complementary workload with conflicting profiles, fair-share schedulers give a false-promise of sharing, but are unable to deal with varying workload in the long run. In scenarios where workloads are simply different, the scheduler assigns resources to the user capable of loading the system at a higher rate relative to other users. This causes temporary starvation of users as shown in cases 1, 3 and 5.

Identifying such problems is the key contribution of the current work while solving them is our future research.

REFERENCES

[1] D. Fotakis, S. Kontogiannis, E. Koutsoupias, M. Mavronicolas, and P. Spirakis, *The structure and complexity of Nash equilibria for a selfish routing game*, in Automata, Languages and Programming, P. Widmayer, S. Eidenbenz, F. Triguero, R. Morales, R. Conejo, and M. Hennessy, eds., Lecture Notes in Comput. Sci., Vol. 2380, Springer, Berlin, 2002, pp. 123–134.
[2] W. R. Stevens, M. Allman, and V. Paxson, TCP *congestion control*, Consultant (1999).
[3] J. Kay and P. Lauder, *A fair share scheduler*, Commun. ACM, 31 (1988), pp. 44–55.
[4] D. Ferraioli and C. Ventre, *On the price of anarchy of restricted job scheduling games*, in ICTCS, 2009, pp. 113–116.
[5] J. F. Nash, *Equilibrium points in n-person games*, Proc. Nat. Acad. Sci. U.S.A., 36 (1950), pp. 48–49.

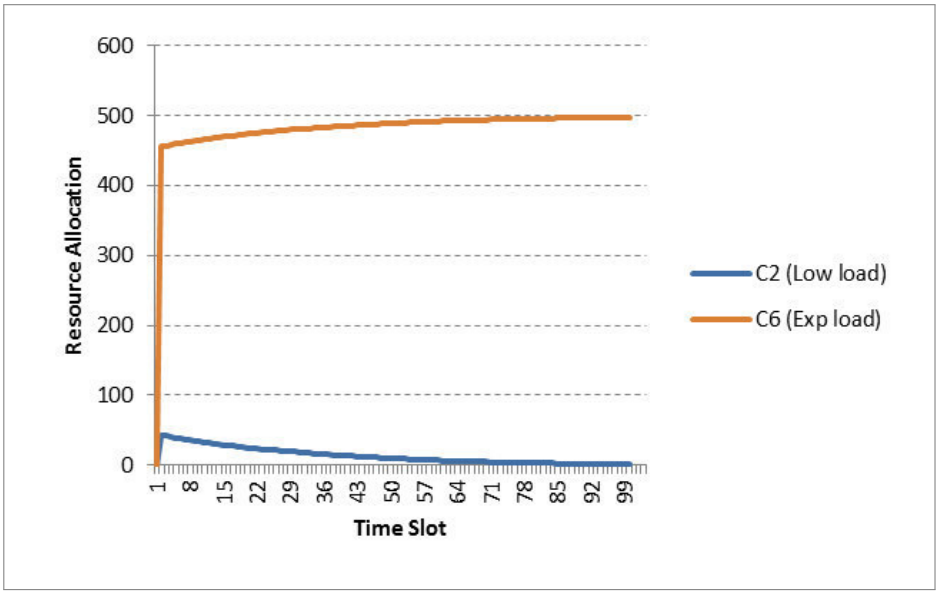

Fig. 6.8: Resource Allocation for Case 3

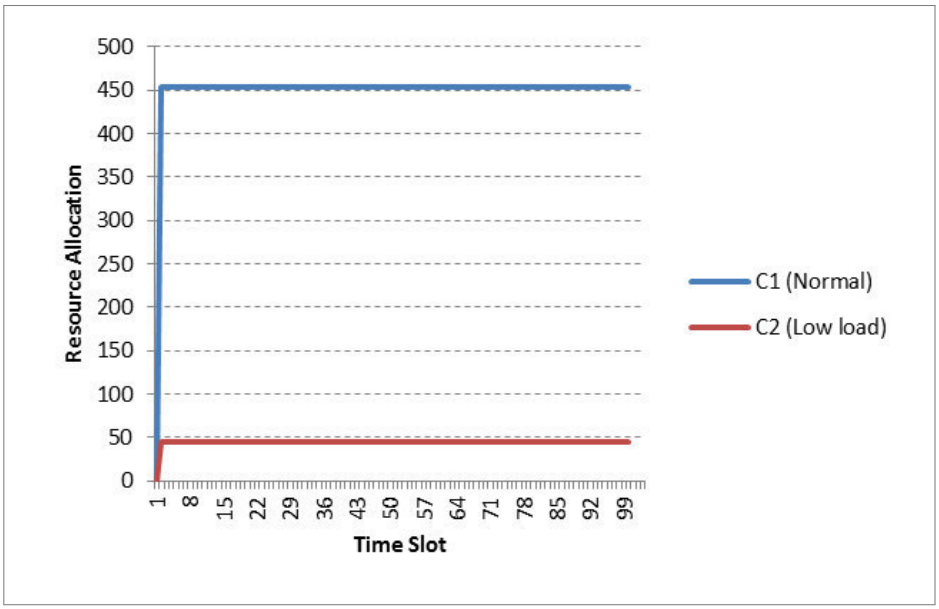

Fig. 6.9: Resource Allocation for Case 4

[6] A. Fiat, H. Kaplan, M. Levy, and S. Olonetsky, *Strong price of anarchy for machine load balancing*, in Automata, Languages and Programming, L. Arge, C. Cachin, T. Jurdziński, and A. Tarlecki, eds., Lecture Notes in Comput. Sci., Vol. 4596, Springer, Berlin, 2007, pp. 583–594.

[7] L. Epstein, M. Feldman, T. Tamir, Ł. Witkowski, and M. Witkowski, *Approximate strong equilibria in job scheduling games with two uniformly related machines*, Discrete Appl. Math., 161 (2013), pp. 1843–1858.

[8] M. Feldman and T. Tamir, *Approximate strong equilibrium in job scheduling games*, J. Artificial Intelligence Res., 36 (2009), pp. 387–414.

[9] C. Daskalakis, *Nash equilibria*: *Complexity, symmetries, and approximation*, Computer Science Review, 3 (2009), pp. 87–

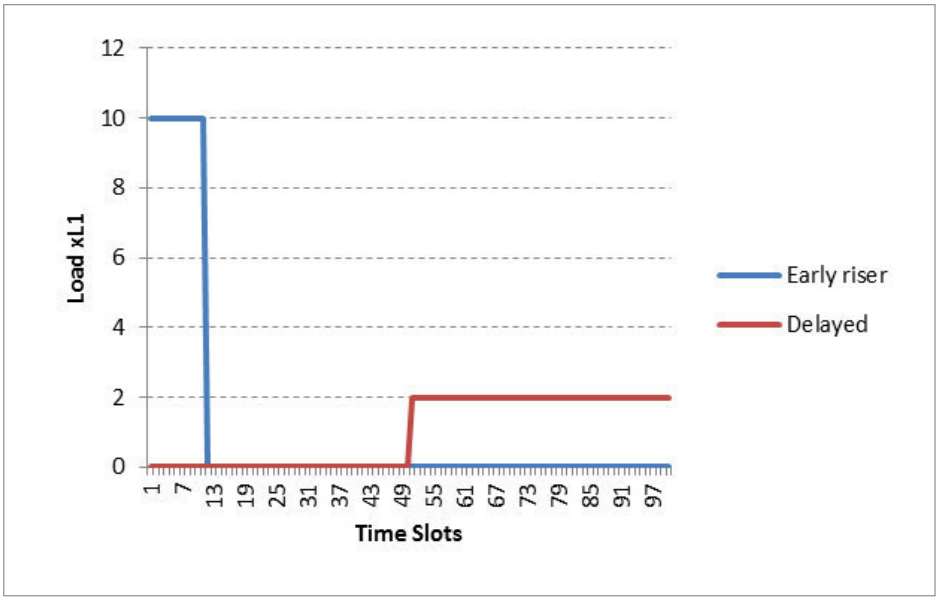

Fig. 6.10: Load Profile for Case 5

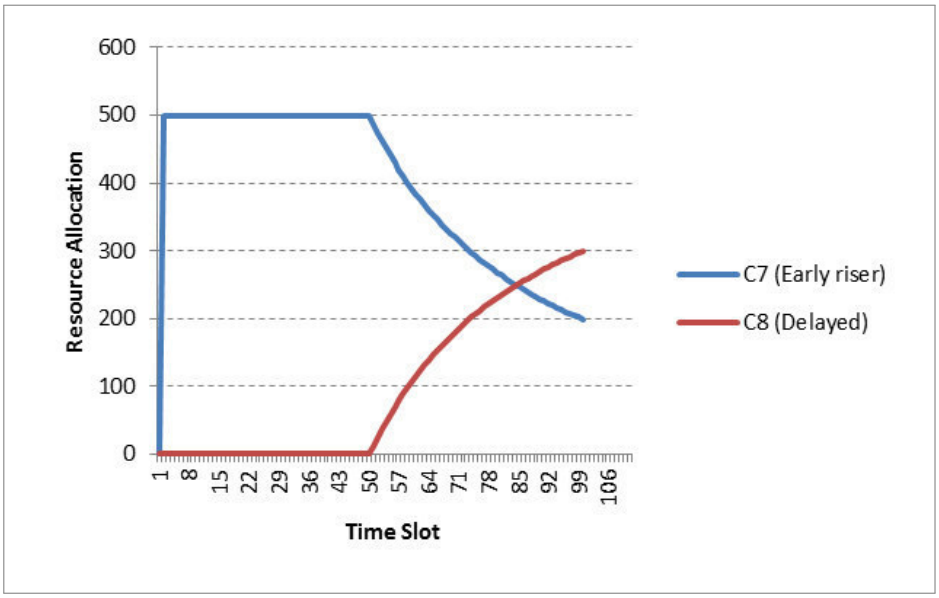

Fig. 6.11: Resource Allocation for Case 5

100.

[10] F. Pinel, J. E. Pecero, S. U. Khan, and P. Bouvry, *Energy-efficient scheduling on milliclusters with performance constraints*, in Proceedings of the 2011 IEEE/ACM International Conference on Green Computing and Communications, IEEE Computer Society, Washington, DC, 2011, pp. 44–49.

[11] L. Wang, S. U. Khan, D. Chen, J. Koodziej, R. Ranjan, C.-z. Xu, and A. Zomaya, *Energy-aware parallel task scheduling in a cluster*, Future Generation Computer Systems, 29 (2013), pp. 1661–1670.

[12] P. Zhang, Y. Gao, J. Fierson, and Y. Deng, *Eigenanalysis-based task mapping on parallel computers with cellular networks*, Math. Comp., 83 (2014), pp. 1727–1756.

[13] P. Zhang and Y. Deng, *Design and analysis of pipelined broadcast algorithms for the all-port interlaced bypass torus networks*, IEEE Trans. Parallel Distrib. Systems, 23 (2012), pp. 2245–2253.
[14] P. Zhang and Y. Deng, *An analysis of the topological properties of the interlaced bypass torus (*iBT*) networks*, Appl. Math. Lett., 25 (2012), pp. 2147–2155.
[15] Y. Deng, P. Zhang, C. Marques, R. Powell, and L. Zhang, *Analysis of Linpack and power efficiencies of the world's* TOP500 *supercomputers*, Parallel Comput., 39 (2013), pp. 271–279.
[16] J. J. Dongarra, H. W. Meuer, and E. Strohmaier, TOP500 *supercomputer sites*, Supercomputer, 13 (1997), pp. 89–120.
[17] P. Zhang, R. Powell, and Y. Deng, *Interlacing bypass rings to torus networks for more efficient networks*, IEEE Trans. Parallel Distrib. Systems, 22 (2011), pp. 287–295.
[18] R. Feng, P. Zhang, and Y. Deng, *Network design considerations for exascale supercomputers*, in Proceedings of the 24th IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 2012), 2012, pp. 86–93.
[19] D. Klusácek, *Scheduling in grid environment*, Ph.D. thesis, Masaryk University, Brno, Czech Republic, 2008.
[20] M. Q. Xu, *Effective metacomputing using* LSF *multicluster*, in Proceedings of the 1st IEEE/ACM International Symposium on Cluster Computing and the Grid, IEEE Computer Society, Washington, DC, 2001, p. 100.
[21] W. Gentzsch, *Sun grid engine*: *Towards creating a compute power grid*, in Proceedings of the 1st IEEE/ACM International Symposium on Cluster Computing and the Grid, IEEE Computer Society, Washington, DC, 2001, pp. 35–36.
[22] H. Feng, V. Misra, and D. Rubenstein, PBS: *a unified priority-based scheduler*, in ACM SIGMETRICS Performance Evaluation Review, 35 (2007), pp. 203–214.
[23] F. Berman, G. Fox, and A. J. Hey, *Grid Computing*: *Making the Global Infrastructure a Reality, Vol. 2*, John Wiley and Sons, New York, 2003.
[24] Adaptive Computing, *Moab* HPC *Suite - Basic Edition* 8.0.0, September 2014.
[25] E. Huedo, R. S. Montero, and I. M. Llorente, *The GridWay framework for adaptive scheduling and execution on grids*, Scalable Computing: Practice and Experience, 6 (2001).
[26] M. Romberg, *Unicore*: *Beyond web-based job-submission*, in Proceedings of the 42nd Cray User Group Conference, 2000, pp. 22–26.
[27] J. Huang, Y. Wang, N. Vaidyanathan, and F. Cao, GRMS: *A global resource management system for distributed* QoS *and criticality support*, in Proceedings of the 1997 International Conference on Multimedia Computing and Systems, IEEE Computer Society, Washington, DC, 1997, pp. 424–432.
[28] M. Harchol-Balter, B. Schroeder, N. Bansal, and M. Agrawal, *Size-based scheduling to improve web performance*, ACM Transactions on Computer Systems, 21 (2003), pp. 207-233.
[29] K. Aida, *Effect of job size characteristics on job scheduling performance*, in Job Scheduling Strategies for Parallel Processing, D. G. Feitelson and L. Rudolph, eds., Lecture Notes in Comput. Sci., Vol. 1911, Springer, Berlin, 2000, pp. 1–17.
[30] S. Baruah, S. Funk, and J. Goossens, *Robustness results concerning* EDF *scheduling upon uniform multiprocessors*, IEEE Trans. Comput., 52 (2003), pp. 1185–1195.
[31] J. Nash, *Non-cooperative games*, Ann. of Math. (2), 54 (1951), pp. 286–295.
[32] V. J. Leung, G. Sabin, and P. Sadayappan, *Parallel job scheduling policies to improve fairness*: *a case study*, in Proceedings of the 2010 39th International Conference on Parallel Processing Workshops, IEEE Computer Society, Washington, DC, 2010, pp. 346–353.
[33] G. Christodoulou, L. Gourves, and F. Pascual, *Scheduling selfish tasks*: *About the performance of truthful algorithms*, in Computing and Combinatorics, G. Lin, ed., Lecture Notes in Comput. Sci., Vol. 4598, Springer, Berlin, 2007, pp. 187–197.
[34] N. Andelman, M. Feldman, and Y. Mansour, *Strong price of anarchy*, in Proceedings of the 18th Annual ACM-SIAM Symposium on Discrete Algorithms, ACM, New York, 2007, pp. 189–198.
[35] S. D. Kleban and S. H. Clearwater, *Fair share on high performance computing systems*: *What does fair really mean?*, in Proceedings of the 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid, IEEE Computer Society, Washington, DC, 2003, pp. 146–153.
[36] H. Hussain, S. U. R. Malik, A. Hameed, S. U. Khan, G. Bickler, N. Min-Allah, M. B. Qureshi, L. Zhang, W. Yongji, and N. Ghani, *A survey on resource allocation in high performance distributed computing systems*, Parallel Comput., 39 (2013), pp. 709–736.
[37] T. L. Casavant and J. G. Kuhl, *A taxonomy of scheduling in general-purpose distributed computing systems*, IEEE Trans. Software Engrg., 14 (1988), pp. 141–154.
[38] H. El-Rewini, T. G. Lewis, and H. H. Ali, *Task Scheduling in Parallel and Distributed Systems*, Prentice-Hall, Inc., Upper Saddle River, NJ, 1994.
[39] H. A. James, *Scheduling in metacomputing systems*, Ph.D. thesis, Department of Computer Science, University of Adelaide, Australia, 1999.
[40] C. Boneti, F. J. Cazorla, R. Gioiosa, A. Buyuktosunoglu, C.-Y. Cher, and M. Valero, *Software-controlled priority characterization of* power5 *processor*, ACM SIGARCH Computer Architecture News, 36 (2008), pp. 415–426.
[41] C. Boneti, R. Gioiosa, F. J. Cazorla, and M. Valero, *A dynamic scheduler for balancing* HPC *applications*, in Proceedings of the 2008 ACM/IEEE Conference on Supercomputing, IEEE Press, Piscataway, NJ, 2008, p. 41:1–41:12.
[42] R. C. Larson, OR *forum-perspectives on queues*: *social justice and the psychology of queueing*, Oper. Res., 35 (1987), pp. 895–905.
[43] R. W. Rosenthal, *A class of games possessing pure-strategy Nash equilibria*, Internat. J. Game Theory, 2 (1973), pp. 65–67.
[44] E. Angel, E. Bampis, and N. Thibault, *Randomized truthful algorithms for scheduling selfish tasks on parallel machines*, in LATIN 2010: Theoretical Informatics, A. López-Ortiz, ed., Lecture Notes in Comput. Sci., Vol. 6034, Springer, Berlin, 2010, pp. 38–48.
[45] E. Angel, E. Bampis, and F. Pascual, *Truthful algorithms for scheduling selfish tasks on parallel machines*, Theoret.

Comput. Sci., 369 (2006), pp. 157–168.

Edited by: Dana Petcu
Received: July 29, 2014
Accepted: Sept 21, 2014