Art Sedighi · Milton Smith

# Fair Scheduling in High Performance Computing Environments

Springer

# Fair Scheduling in High Performance Computing Environments

Art Sedighi • Milton Smith

# Fair Scheduling in High Performance Computing Environments

Springer

Art Sedighi
Industrial, Manufacturing
& Systems Engineering
Texas Tech University
Lubbock, TX, USA

Milton Smith
Industrial, Manufacturing
& Systems Engineering
Texas Tech University
Lubbock, TX, USA

ISBN 978-3-030-14567-5 ISBN 978-3-030-14568-2 (eBook)
https://doi.org/10.1007/978-3-030-14568-2

Library of Congress Control Number: 2019934954

# Preface

This book introduces a new scheduler—the Rawlsian Fair scheduler—which can distribute resources fairly in situations in which users with different usage profiles are competing for resources in a large, shared computing environment. The Rawlsian Fair scheduler is demonstrated to increase performance and reduce delay in high-performance computing workloads of four classes:

Class A – Similar but complementary workloads
Class B – Steady vs. intermittent workloads
Class C – Large vs. small workloads
Class D – Large vs. noise-like workloads

The Rawlsian Fair scheduler achieves short-term fairness in cases in which varying workloads and usage profiles require rapid responses. It is shown to consistently benefit workloads in C and D situations and to benefit workloads of disproportionate sizes in A and B situations.

This dissertation also presents a new simulation framework—dSim—which was created to simulate the new Rawlsian Fair scheduler. A series of simulations performed using dSim demonstrate that the Rawlsian Fair scheduler makes scheduling decisions that can ensure instantaneous fairness in high-performance computing environments. Because it does so, the Rawlsian Fair scheduler can both maximize user satisfaction and ensure that computational resources are utilized efficiently.

Lubbock, TX, USA

Art Sedighi
Milton Smith

# Acknowledgments

This effort would not have been possible without the help of Dr. Milton Smith. I offer special thanks to Dr. Smith for his patience throughout our many rounds of discussion of my raw ideas and for helping me to sort out my thoughts.

No words can describe my gratitude to Dr. Deng. For many years, you had to listen to me talk about scheduling, fairness, gameplay, etc. Thank you for your support and help through this long, and at times unfruitful, process.

Dr. Mario, you are an inspiration and a jewel of Texas Tech and the academic world in general. I cannot think of anyone else who can handle 2 weeks of 12-hour days in the same room and still inspire the audience from the first moment to the last. I will always carry your words of wisdom with me.

I also extend my thanks to Dr. Burns and Dr. Du for affirming my thoughts and making me think about scenarios I would not otherwise have thought of. I am proud to have you both as members of my committee.

Irina, my wonderful wife, thank you for your love, support and patience. Thank you for allowing me to pursue my dreams. I know this process has been long and, at times, very taxing on our life, so I want to thank you for your love and understanding. I love you very much. Special thanks also go to my boys, Isaac, Benjamin, and Asher. I love you. I hope that you will someday have the opportunity to pursue your own dreams. I also want to thank my parents for teaching me the importance of education and to thank Aylin, my sister, for simply being who she is!

I also want to thank Peng Zhang—I truly enjoyed our discussions—and my cousin, Dan Zadok, for spending many hours with me during our SoftModule days discussing the merits of scheduling theory.

# Contents

# Chapter 1
# Introduction

## 1.1 Background

While some industries use High Performance Computing (HPC) to increase profits, other industries require HPC to do business at all. In the oil and gas industries, HPC is used to conduct seismic simulations. In the pharmaceutical industry, HPC is used to discover new drugs.

In finance, the calculation of risk is an essential part of doing business (Gleeson 2010; Hakenes and Schnabel 2011; McNeil et al. 2015; Tarullo 2008). Since the 2008 financial crisis (Demirguc-Kunt et al. 2013; Elliott 2009; White 2008, 2009), governmental regulations, such as BASEL III (Committee 2010), have required each bank to know the current status of their portfolio, how their portfolio could be affected by various market conditions, and decision making process that went into making their own transactions. These restrictions enable banks and other financial institutions to assess risk quickly.

The primary way to fulfill requirements set forth by regulatory bodies it to simulate various possibilities and future outcomes. Monte Carlo Simulations (MCS) are the primary method that financial institutions use to calculate risk (Glasserman et al. 2010; Reyes et al. 2001; Tezuka et al. 2005). MCS jobs are parallelizable and can be used to speed calculations in HPC systems. An MCS job run in an HPC system is decomposed into an array of smaller tasks, each of which is submitted and executed separately. These tasks can be of various sizes and may require a variety of system configurations. In agreement with Feitelson, Rudolph, Schwiegelshohn, Sevcik, and Wong (1997), our work assumes that these tasks are malleable and, therefore, that additional resources reduce the execution time of a given job.

Since federal regulations like BASEL III (Committee 2010) require banks to compile information that gives them the ability to calculate risk more quickly and, therefore, to conduct more business, the need for computational power has increased. Historically, however, datacenter utilization has remained between 5%

A. Sedighi, M. Smith, *Fair Scheduling in High Performance Computing Environments*, https://doi.org/10.1007/978-3-030-14568-2_1

and 20% on average (Armbrust et al. 2010). This underutilization is especially problematic because when management and maintenance costs are taken into consideration, the cost of purchasing a computer is only about a third of the total cost of operating that computer for 3 years (Marston et al. 2011). About 80% of this total cost is wasted when the computer is not fully utilized.

To cut costs and raise output[1] (Weisbord 2004), a scientific management approach is required (Taylor 1911). The primary cost is caused by the need for high computational requirements (i.e. servers). To reduce cost, it is best to use resources as efficiently as possible. The wasting of computational resources can be minimized by sharing resources among many users, and a scheduler is a mediator that controls access to such resources to ensure that they are utilized *fairly*.

Scheduling tasks that utilize a set of shared resources is not without its challenges, however. For one, the fact that users must share access with one another inevitably generates competition for scarce computational resources.

To avoid such competition schedulers work primary to increase utilization and to reduce "time-in-system"—the time it takes for a request to be processed (Baker and Trietsch 2013). All schedulers should be axiomatically fair (Kay and Lauder 1988), i.e. appear and seem fair, while optimizing for two system parameters. These two parameters are: "task load requirement" and "task priority"[2] (Parker 1996).

"Fairness" is not well defined in the literature, however. In scheduling systems, it is stipulated that schedulers need only to seem fair, and this has translated into long-term fairness across all users and all tasks (Kay and Lauder 1988). As time-in-system is not a variable to be optimized, some users might experience undesirable temporal starvation. In essence, short-term fairness is occasionally sacrificed for long-term equality (Bertsimas et al. 2011). This was demonstrated by showing how disproportionate users can be temporarily starved and delayed in systems that employ a fair-share scheduling mechanisms (Sedighi et al. 2014).

In addition, current fair-share algorithms assume that task submission follows a Poisson distribution (Kay and Lauder 1988; Kleban and Clearwater 2003), which may not always be the case—for instance, when a burst of incoming tasks caused by a market event requires an instantaneous response.

## 1.2 Problem Statement and Scope

In cases where quick decisions is required, long-term fairness is not acceptable because the business opportunity may not exist. Such situations are usually seen in financial institutions to keep up with the rapid changes in the global financial markets. Small risk-modeling simulations—such as those submitted by trading

[1](Weisbord 2004) p.81 is based on Lewin's "Humanization of the Taylor System: An Inquiry into the Fundamental Psychology of Work and Vocation" published in 1920.

[2]See Sect 1.7

desks—can compete with large, bank-level risk-modeling simulations. Both small and large jobs are important (i.e. they both have "priority"), however, and because MCS simulations are broken down into thousands or even millions of smaller tasks (i.e. "load"), it is nearly impossible for fair-share schedulers to determine the order in which such tasks should be scheduled. As mentioned in the previous section, conventional schedulers—outlined by Baker and Trietsch (2013)—make decisions using two primary parameters: load and priority. A third parameter that can be used is "seniority," which can be dynamically updated and indicates the time a task has spent in the system.

Multi-criteria (or "composite") schedulers (Hoogeveen 2005; T'kindt and Billaut 2006) make decisions using multiple parameters. When an HPC environment contains a large number of short-running tasks, seniority can be used to ensure that no user is temporarily starved. This variable is additive, increasing the longer the task is in the system.

It can assumed that the task-submission profile is unknown and does not follow a Poisson distribution (Harchol-Balter 2013) and that schedulers must be able to deal with varying sets of workloads.

### *1.2.1 Class A – Complementary Intermittent Workloads*

In this type of workload, users send tasks to be executed in an intermittent fashion, in a way that there are complementary users. Complementary users are two or more users who send tasks when the other(s) are not. In mathematical terms, one user's submission pattern resembles a sine wave, and the other user's submission pattern resembles a cosine wave.

### *1.2.2 Class B – Steady vs. Intermittent Workloads*

This type of workload resembles a long-running workload versus an intermittent workload. A long-running workload is a job that is divided into many smaller tasks that are submitted at a steady rate. The submission rate of a long-running workload can be limited by the bandwidth between the user and the HPC environment, the processing power of the user, or any number of other variables. In an intermittent workload, a user or a set of users submits tasks in a pattern that resembles a sine wave or a cosine wave.

### *1.2.3 Class C – Large vs. Small Transient Workloads*

In this type of workload, a user’s small workload is forced to compete for resources with much larger workloads. This user has a one-time submission with a disproportionately small task count, while the other users in the system have much larger usage requirements.

### *1.2.4 Class D – Large vs. Noise-Like Workloads*

In this workload, one user’s workload is so small relative to another’s that from the perspective of the larger workload, it appears as noise. The rate of submission, though very small, may be consistent and require attention.

## 1.3 What Is Covered in This Book

This book will introduce a new methodology that can provide an alternative solution to the current fairshare scheduling and by doing do reduce or eliminate the temporal starvation that is caused by current schedulers that tend to benefit larger users over smaller ones. This new scheduling algorithm is used in high performance computing environment and is based on the Rawlsian definition of justice and fairness. Rawlsian fairness utilizes on the concept of the Least Well-off User which gives precedence to certain type of tasks seen in HPC environments.

A new parameter, called seniority, is introduced in this research to monitor and control how a scheduler allocates resources to the Least Well-off User.

## 1.4 Seek and You Shall Find

Scheduling systems sometimes allocate resources unfairly to users whose workloads do not fit a Poisson distribution. Although fair-share scheduling can achieve long-term fairness, it does so by sacrificing temporal fairness to some users.

The primary question that this book seeks to answer is whether a multi-criteria scheduler with the capacity for adaptive aging can overcome the unfairness endemic to other fair-share schedulers. The hypothesis of this work is that an HPC scheduler that uses a tertiary parameter—time-in-system—can reduce the task-execution delay experienced by users with disproportionate usage profiles.

In order to find the answers to the above, the following sub-questions must be addressed:

1. Is there a multi-criteria scheduling algorithm capable of accounting for seniority—i.e. the time each task has spent in the system?
2. Is there a model that can represent workloads that are transiently non-Poisson distributed? Can such a model be used to represent a scheduler with multiple optimization variables—e.g. load, priority, and seniority?
3. Is there a model for "fairness factor," a multiplier that can alter the position of a task awaiting execution?
4. How can a model for fairness factor be verified through simulation? Can a simulator be created that is capable of representing HPC environments with various types of users and workloads and of predicting the performance of any user-developed scheduling algorithm?
5. Is it possible to prove that the newly developed scheduler is quantitatively fair—specifically, that its decisions accord with the rules of utility and fairness (as outlined by Nash's fairness model, which is based on bargaining theory) and maximize the least utility (as is mandated by Rawls)?
6. How would such a scheduler deal with disproportionate job profiles, in which smaller workloads are generally neglected in favor of larger ones?

## 1.5 Expected Outcome

Our work has three objectives:

1. The primary objective is to evaluate the efficacy of a scheduler that uses three criteria: load, priority, and seniority. The scheduler build for this research can reduce the time-in-system of a task by considering the task's seniority as well as its load and priority.
2. The second objective is to quantitatively demonstrate through simulation how fairness alters the performance characteristics of the new scheduler.
3. The third objective is to demonstrate the capabilities of a simulation platform that can be used to test user-supplied scheduling algorithms.

# Chapter 2
# Financial Market Risk

In the first 5 min (9:30–9:35 AM EST) after the market opened on Friday, June 24, 2016, the trading volume of the Dow Jones Industrial Average reached 5.71 million shares; by the closing minute (4:00 PM), the volume was over 63 million shares[1] (Table 2.1). Over the course of the day, a total of 5.2 million trades were processed by the New York Stock Exchange (NYSE), and over five million of these were small trades of 1–2000 shares.[2]

Many trades are automatic and entered using complex, algorithm-driven programs. This phenomenon is known as "algorithmic trading" or "algo-trading." Financial markets run Value-at-Risk (VaR) models to assess the risk and potential reward of each trade (Alexander 1998; Jorion 1997). The speed at which a VaR model runs determines the rate at which a given algo-trading application can enter trades (Chong et al. 2009).

VaR models are based on Monte Carlo Simulations (MCSs) (Reyes et al. 2001), and are constituted by both simple and complex mathematical functions that approximate the probability of an outcome by simulating trials featuring one or more variables and a specified time horizon (Krishnamurthy et al. 2012). Because of their sheer size, these models require extensive computational power and, as a result, can be significantly parallelized across High Performance Computing (HPC) environments.

The accuracy of a Monte Carlo Simulation depends on its number of runs (i.e. "paths"), and varying its input parameters (known as "shocking" the simulation) can yield a clearer picture of potential outcomes (Glasserman et al. 2010). For example, a financial institution may want to calculate the VaR of a stock portfolio by running a pricing calculation with 1,000,000 paths and shocking the simulation by varying the interest rate, gold price, and dollar exchange rate. If each of the

[1]Data were gathered using http://finance.yahoo.com/echarts?s=%5EDJI+Interactive#{"range": "1d","allowChartStacking":true}

[2]Data were taken directly from the NYSE website on June 24th, 2016. Data can be retrieved from: http://www.nyxdata.com/Data-Products/NYSE-Volume-Summary#summaries

A. Sedighi, M. Smith, *Fair Scheduling in High Performance Computing Environments*, https://doi.org/10.1007/978-3-030-14568-2_2

**Table 2.1** Per-minute shares-traded volume on the NYSE for June 24, 2016

| Time | Number of shares |
|---|---|
| 9:30 AM | 1.67 million |
| 9:31 AM | (no data – Potentially summed up in the next minutes) |
| 9:32 AM | 2.84 million |
| 9:33 AM | 633 K |
| 9:34 AM | 598 K |
| 9:35 AM | 996 K |
| | : |
| 3:55 PM | 1.57 million |
| 3:56 PM | 1.58 million |
| 3:57 PM | 2.16 million |
| 3:58 PM | 2.65 million |
| 3:59 PM | 5.96 million |
| 4:00 PM | 63.32 million |

shocks affects only two values (e.g. the current market value and one step up or down[3]), the MCS needs to run 1,000,000 paths * 2 (for the interest-rate shock) * 2 (for the price-of-gold shock) * 2 (for the dollar-exchange-rate shock). The paths in this scenario total six million, and each predicts the value of the portfolio at some future time—e.g. 30 days out or 5 years out. This case is very simple, featuring only 3 shocks with 2 values each. Significantly greater numbers of shocks with larger numbers of potential values are feasible, however, and in such cases, predicting the potential outcome is an unlikely event (Glasserman et al. 2010). Moreover, as the above example suggests, more complex models that generate greater numbers of paths require significantly more computing power.

The size and complexity of Monte Carlo Simulations make them great candidates for HPC environments. MCS models can be broken into large numbers of malleable tasks, which are packaged into a job. Regardless of the user or the MCS job, these tasks are similar in both size and required computational power. In contrast, classical, job-based scheduling methodologies such as Shortest Processing Time (Baker and Trietsch 2013) and Earlier Due Date (Parker 1996) do not apply because the tasks resemble one another and the collection of completed tasks constitutes the desired result of the job. The tasks are not submitted at the same time, and the scheduler does not know the number of tasks still pending. A challenge thus arises when there is a vast discrepancy between two or more users running their models at the same time: the users compete for additional resources to finish more quickly, and the scheduler is required to allocate resources fairly.

In cases in which an infinite number of tasks is incoming for a particular job, it has been demonstrated that schedulers temporarily starve users with fewer pending tasks (Sedighi et al. 2014). The tasks are queued in the scheduler. To empty the queue as

[3]For example, if the current interest rate is 2.5% and interest rate changes by.25% at a time, one step up would be 2.75%, and one step down would be 2.25%.

fast as possible, fair-share schedulers (Bach 1986a, b; Kay and Lauder 1988) assign increased precedence to jobs with larger pending tasks in the queue. This can temporarily starve jobs with fewer pending tasks.

Our work considers a workload that is not Poisson-distributed because the requests that have entered the system are related and close together. When all of the tasks constituting a given job have been completed, the job is finished. Then, the next job (which may or may not be related to the previous job) enters the system.

Particular conditions in the financial market cause bursts of jobs to enter the system. When a burst of jobs enters the system, the jobs use the environment and then leave. Whether and when a given job will return is unknown. In addition, a given job can be composed of two tasks: one at the beginning of the day ("market open") and 1 h later, at the end of the day ("market close"). Because all of the tasks that constitute a given job are related, a job cannot be finished until all of its tasks are completed. The user determines when the job is finished; the scheduler is unaware of "job start" and "job finish" events.

## 2.1 Relevance

The scenario described in the previous section lends itself in various ways to the problem statement vis-a-vis the different classes of workloads (Table 2.2).

**Table 2.2** Relevance of workload types to problem statement

| Workload classification | Relevance |
|---|---|
| Class A | Intermittent workloads occur when users submit jobs throughout the day. Examples of such jobs include requests for risk calculations submitted by traders under normal market conditions. Such jobs may or may not be complementary to other jobs submitted. |
| Class B | Class B workloads occur when a steady workload and an intermittent workload compete for resources. For example, a firm-level risk calculation might compete with an individual trader's risk request. The firm-level workload is large and arrives continuously, and it is limited by external constraints, including the bandwidth of the network, the power of the CPU, etc. In contrast, the trader's workload represents the trader's own actions and potential future actions, and it is intermittent because it varies with market conditions. |
| Class C | Competing for resources for a one-time burst submission and larger workloads represents cases with smaller trades amongst the sea of much larger trades from various traders. Another example is an end-of-day request, in which a trader needs to calculate the value of their portfolio and thus sends a one-time request for a tally of their position. |
| Class D | Reporting and other forms of general verification occur throughout the day and can be considered background noise. These jobs may be very small—for instance, requests to sum-up the positions since the last request—but may be immensely valuable from a book-keeping perspective. |

Chapter 8 will cover simulations designed to compare the efficacy of a new scheduling paradigm to that of fair-share in handling these four types of workloads.

## 2.2 An Example

This section offers an example in which temporal starvation occurs because two users compete for resources in an HPC environment. While the number of resources is the same and does not change for both users, the number of tasks and the pattern in which the tasks are submitted vary between the users. This two-user case can easily be extended to incorporate *n* users, where users *1..k-1* compete for resources against user k.

$$U_k = \begin{cases} x, number\ of\ Tasks(k) \leq \sum_{i=1}^{k-1} \#\ of\ Tasks(i) \\ < x, number\ of\ Tasks(k) \ll \sum_{i=1}^{k-1} \#\ of\ Tasks(i) \end{cases} \tag{2.1}$$

where $U_k$ is the level of satisfaction or utility experienced by user k. Equation 2.1 states simply that a user can experience different levels of satisfaction ($x$ or $<x$) not only because of their own submission profile ($Tasks(k)$), but also because of the actions of others ($Tasks(1..\ k - 1)$). If the number of tasks submitted by k is much smaller than the total number of tasks submitted by the other users, then the utility experienced by k (denoted "$U_k$") will be less than it would have been had k's total number of tasks more closely matched the number of tasks submitted by the other users. This is the case because current fair-share schedulers normalize requests, achieving long-term fairness at the expense of short-term fairness to small users. As a result, utility is a function not only of one's own submission profile, but also of the collective's submission profile (see Sect. 4.2) (Fig 2.1).

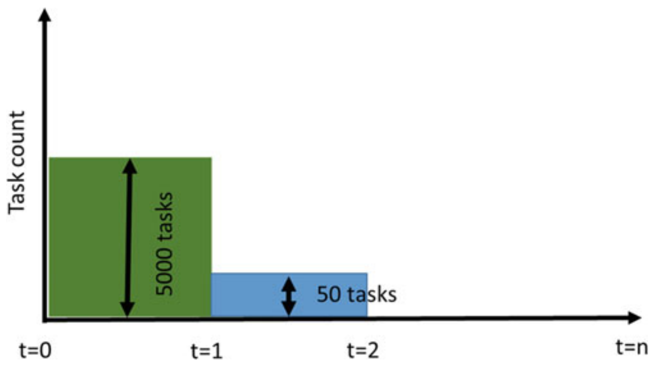


**Fig. 2.1** Comparison of the task submissions for Job A (green) and Job B (blue)

**Table 2.3** Parameters for the two-user example

| Jobs | Load in $\frac{\text{tasks}}{\text{time slice (PC)}}$ |
|---|---|
| A | $\mathrm{L}_1(\mathrm{x}, \mathrm{PC}) = \max\{5000^* \mathrm{L}_0(\mathrm{x}, \mathrm{PC}) \cos(\pi^* \mathrm{PC}/2), 0\}, \sum_{PC=0}^{n} L_1(PC) = 5000$ |
| B | $\mathrm{L}_2(\mathrm{x}, \mathrm{PC}) = \max\{50 * \mathrm{L}_0(\mathrm{x}, \mathrm{PC}) \sin(\pi^* \mathrm{PC}/2), 0\}, \sum_{PC=0}^{n} L_2(PC) = 50$ |

Consider a system with only two jobs—A and B—which have the following characteristics:

– Job A has 5000 tasks in total. All of these tasks are entered at the start time for the simulation, t = 0. All of these tasks are also of equal load (L) and priority (P).
– Job B has 50 tasks in total. All of these tasks are entered at the start time for the simulation, t = 1. All of these tasks are also of equal load (L) and priority (P), and are the same as those of Job A.

The submission profile for this example is shown in Table 2.3, in which A sends a total of 5000 tasks and B sends a total of 50 tasks. Both users submit their tasks at once.

In Table 2.3, "PC" is the placement counter, a clock tick of the system. "Task" is represented by x, and $L_0$ (x, PC) is the base task load of 1 task/PC.

An HPC system is composed of a scheduler and a set of servers or resources that do the computation. A fair-share scheduler mediates access to the backend resource pool of 100 computers that make up the HPC system. Each task takes 1 unit of time to complete.

When Job A is the only job in the system, job A would take 5000/100 (or 50) units of time to complete Job A. 100 tasks are dispatched at one at a time by the scheduler to the available resources, and since the processing time is constant, all tasks finish at the same amount of time. As soon as a batch is finished, the next batch of 100 tasks is scheduled. As a result, Job A alone takes 50 scheduling cycles to complete.

When Job B is the only job in the system, one 1 unit of time is needed to complete Job B.

Because each task takes 1 unit of time to finish and there are more available resources than there are tasks in job B, the tasks in Job B are parallelized across the resources and completed in one unit of time. When the two users are forced to share the system, however, the scheduler distributes the tasks in proportion to the total number of tasks pending in the queue. Thus, Job A gets $\frac{5000}{5050} * 100 \sim 99$, and Job B gets the remaining resource of 1. In the next round, Job A gets $\frac{4901}{4950} * 100 \sim 99$, and so on. This is progression is illustrated in Fig. 2.2.

If the sizes of Job A and Job B were known in advance, the scheduler could employ the Shortest-Job-First algorithm and schedule Job B first. Since the scheduler does not know when all of the requests have been submitted for a given job, a Poisson process property, it cannot reserve resources nor make a decision as to what

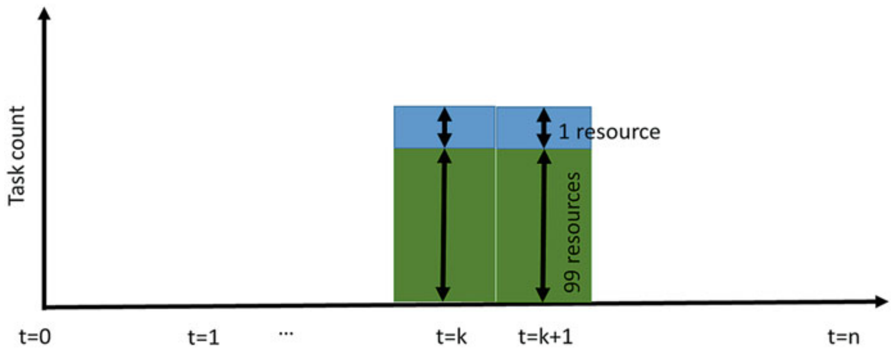


**Fig. 2.2** The distribution of resources by a fair-share scheduler to Job A (green) and Job B (blue)

**Table 2.4** Scheduling results of Job A and Job B

| Job name | Scheduling sequence | Start time | Finish time | Delay from optimal |
|---|---|---|---|---|
| Job A | A goes first | t = 0 | t = 5000/100 = 50 | 0 |
| Job B | A goes first | t = 51 | t = 51 | 50 |
| Job A | B goes first | t = 1 | t = 51 | 1 |
| Job B | B goes first | t = 0 | t = 0 | 0 |
| Job A | Fair-share | t = 0 | t = 51 (estimate) | 1 |
| Job B | Fair-share | t = 1 | t = 51 (estimate) | 50 |
| Job A | Fair-share – simulated | t = 0 | t = 61 (actual) | 11 |
| Job B | Fair-share – simulated | t = 1 | t = 60 (actual) | 59 |

the best method of scheduling would be. A fair-share scheduler's real-time decisions assume that the incoming load is Poisson distributed, but between t = 0 and t = 2, the task submission profile does not follow a Poisson distribution. Instead, it represents a short burst of tasks submitted in close proximity.

As Table 2.4 shows, under fair-share scheduling, both jobs finish at essentially the same time. While Job A is allocated the majority of the resources, Job B is not starved—just delayed. The simulated models (see Chap. 6) considers both communication costs and the wait time in the queue, thereby providing a more realistic estimate of the delay experienced by each job. In either scenario, both jobs finish their runs.[4] Moreover, if we evaluate the HPC system on the long term (t = infinity), both jobs are treated fairly. The temporal starvation experienced by Job B is undesirable, however, especially in cases in which time is short and every second counts.

The value that is missing in determining fairness is the time that a given job has spent in the system. After Job B has been in the system for some time, for example, the priority of its remaining tasks should increase. A new fair-share scheduling system takes into account the seniority of each job in determining which set of

[4]This is referred to as "equifinality" (Bertalanffy 1969)

requests to fulfill next. A determination of seniority values that change over time, and not a value that is supplied as the property of the job, can help solve the problem. The scheduling system assigns a seniority value to the tasks pending for a given job, thereby preventing short-term starvation. A task's seniority is calculated based on its time in the system relative to the other tasks, and this value determines its position in the queue. This value changes, however: if a job changes its submission profile, its seniority value and place in the queue also change.

## 2.3 Expected Outcome

The expected outcome of our work is the development and modeling of a new scheduling algorithm that will be able to reduce the otherwise disproportionate delay experienced by users. To assign such users more advantageous seniority values, the scheduler will employ various criteria, including time-in-system, load, priority, and the states of the other users. As a result, seniority will aid the scheduler in making real-time decisions that will change with the values of various system parameters, including number of users, pending tasks, and access rights.

## 2.4 Definitions

This section provides operational definitions of some of the terms used in our work.

### *2.4.1 High Performance Computing*

A High Performance Computing (HPC) environment is a system of interconnected computers that increase the computational capacity available by providing parallelized execution paths across the pool of available resources. HPC environments are typically shared by a number of users, each of whom require extensive computational power. The resources or the number of servers in an HPC environment are finite (generally on the order of 500–1000 servers) and homogeneous; they usually share memory and include a number of processors in each node. This setup enables two or more unrelated applications to run side-by-side on the same machine but in different process spaces, containers (Soltesz et al. 2007), or even virtual machines.

### 2.4.2 Scheduling

When the demand for computational resources exceeds the available supply, scheduling sequences the tasks to be completed (Conway et al. 2012). Scheduling balances and optimizes three primary parameters: “fairness,” “utilization,” and “dynamicity” (FUD) (Sedighi et al. 2017b).

Because schedulers cannot optimize all three of these parameters simultaneously, one parameter must suffer for the benefit of the other two. Many scheduling systems reduce fairness in favor of utilization and dynamicity (Sedighi et al. 2017a)

### 2.4.3 Task Load (L)

Task load (L) is traditionally measured in CPU cycles per second, but it can also easily be measured in floating-point operations per second (FLOPS) or Giga-FLOPS. Our work assumes that the load of a given task is either given or calculated, that it does not change across the lifetime of the task, and that the task loads of a given job can vary.

### 2.4.4 Priority of the Task (P)

A task’s priority is static and generally user-defined, and it indicates the importance of the task. Priority values vary from task to task and may change. Priority is assigned by the user and may not represent the priority of the task holistically across all of the users and tasks in the system. In essence, priority may be affected by users’ selfish behavior (Angel et al. 2006).

### 2.4.5 Task Seniority (S)

Task seniority is a dynamic, time-dependent value that indicates how long a task has been in the system relative to the current state of the system and to the other users currently in the system. Seniority changes with changes in the flux of the system, i.e. the rate at which tasks are flowing through the system.

### *2.4.6 Time-in-System*

A task's time-in-system—or "flowtime" (Baker and Trietsch 2013)—is the total time it spends in the system, from the time it is submitted to the time at which the result is sent to the user. Although all scheduling algorithms share the goal of reducing the flowtime of every task, each scheduling algorithm pursues this goal differently, depending on its workload and performance requirements.

### *2.4.7 Utility*

Utility is a measure of satisfaction or happiness (Luce and Raiffa 2012). There are two approaches to defining utility: "cardinal" (Bentham 1879), on which utility is considered measurable and comparable, and "ordinal" (Pareto 1919), on which utility is used merely to exert user preferences in choosing an option. This dissertation assumes the cardinal view of utility because the scope of how utility is calculated is defined and boxed to a very specific problem. According to the FUD hypothesis (Sedighi et al. 2017b), fairness is an aspect of scheduling that directly affects the utility of the user.

## 2.5 Book Organization

Chapter 3 offers a review of the landscape of scheduling in high performance computing environments. Chapter 4 delves into the aspects of fairness in scheduling and introduces the concepts of FUD as it pertains to scheduling. Chapter 5 will introduce a new mathematical model for scheduling that will introduce a tertiary parameter, called Seniority into the model. Chapter 6 onwards focuses on the simulation and results of our new scheduler based on the aforementioned tertiary parameter. A simulator that can analyze the performance and characteristics of the scheduler (dSim) is introduced in Chap. 7. In Chap 8, dSim is used to run a number of different scenarios, the data from which are then analyzed in Chaps. 9–13. The analysis focuses on the treatment of varying workload types by the scheduler, rather than on optimizing a single parameter (e.g. time-in-system).

# Chapter 3
# Scheduling in High Performance Computing

## 3.1 Introduction

This chapter provides background information relevant to our objectives:

i. To evaluate the efficacy of a multi-criteria scheduler that uses seniority as well as load and priority to make scheduling decisions.
ii. To demonstrate how fairness affects the performance characteristics this scheduler.
iii. To build and demonstrate the capabilities of a custom simulation platform.

The literature review is divided into the following sections:

i. Scheduling and scheduling theory
ii. High Performance Computing and shared computing
iii. Fair-share scheduling in a HPC environment
iv. Fairness and utility

Each section will review the literature relevant to its specified topic(s), and any gaps in the research will be noted.

## 3.2 Scheduling and Scheduling Theory

General scheduling algorithms are among the intractable problems of computer science and are considered NP-Hard[1] (P Brucker and Knust 2012). As a result, only heuristic methods can be used to solve scheduling problems. While the only exceptions to this rule have historically been solutions to single- and double-machine problems, a polynomial-time algorithm has been developed that can solve the three-

[1]See p.24

A. Sedighi, M. Smith, *Fair Scheduling in High Performance Computing Environments*, https://doi.org/10.1007/978-3-030-14568-2_3

machine scheduling problem in very limited cases (Baker and Trietsch 2013).[2] Scheduling problems combine two problems into one: resource allocation and sequencing. Because resource allocation is unnecessary in the single-machine problem and trivial in the two-machine problem, in which only one of the two machines is used to make a given decision, only sequencing needs to be solved.

Paradoxically, however, the single-machine problem sheds some light on how scheduling should be done: it suggests that there is an absolute and optimal solution. A number of methods have been used to solve the single-machine sequencing problem (Baker and Trietsch 2013)[3]:

– The Adjacent Pairwise Interchange method (Potts and van Wassenhove 1991) two adjacent jobs are compared and interchanged to be in proper ordering (this method is similar to bubble sort).
– Dynamic Programming ( Brucker and Knust 2012) uses the optimality principle and the optimality of sub-problems to make decisions.
– Branch and Bound (Brucker et al. 1994) partitions large problems into smaller sub-problems and calculates the bound for the solution to each sub-problem.
– Dominance Properties reduces the problem space to jobs that can neither be preempted nor allowed to have inserted idle time.

In addition to these optimization methods, a number of heuristic methods (Baker and Trietsch 2013)[4] can be used to sequence jobs to maximize the flow time and minimize the makespan. These methods include genetic algorithms, simulated annealing, greedy search, and taboo search. However, the applicability of the single-machine problem is limited in cases that rely on High Performance Computing, which are the focus of this research.

Scheduling methodologies generally fall into the following basic categories: Flow Shop Scheduling (Baker and Trietsch 2013),[5] Job Shop Scheduling (Baker and Trietsch 2013),[6] and Open Shop Scheduling (Parker 1996).

## 3.2.1 *Flow Shop Scheduling*

In flow shop scheduling (Baker and Trietsch 2013), a product is assembled by machines in a particular series. A good example of flow shop scheduling is bicycle assembly, in which a bicycle is assembled piece by piece by machines to which it is sent in a particular order. The Flow Shop Scheduling Problem closely resembles the Traveling Salesman Problem (TSP), which is known to be intractable in most cases.

[2]See p.236

[3]See p.34–53

[4]See p.57–75

[5]See p.225

[6]See p.325

Flow shop problems have been solved in polynomial time (P), in which up to two (or, in some conditions, three) machines are used (Baker and Trietsch 2013).[7]

### 3.2.2 *Job Shop Scheduling*

In Job Shop Scheduling (Baker and Trietsch 2013), a product is manufactured by multiple machines in no particular order. Paint manufacturing is an example of Job Shop Scheduling: a desired color is made by mixing a number of other colors, but the order in which each color is added to the mixture is unimportant. This scheduling problem is also a case of TSP, albeit a less restrictive one.

### 3.2.3 *Open Shop Scheduling*

An example of Open Shop Scheduling (Parker 1996) is the check-out line at the grocery store: excluding special types of lanes (express lanes, self-checkout, etc.) any cashier can serve any customer. There is a finite set of types of work—checking out and paying—that all cashiers are capable of handling. If a scheduler were to decide where the next customer should go, its decision would be a simple one: it would simply send the next customer to the next available cashier. In this scenario, there is no flow and no dependency. The Open Shop Scheduling problem is unsolvable in general cases, but it can be solved in Polynomial time (P) when there are at most two—or, under certain conditions, three—machines (Parker 1996).[8] Open Shop Scheduling and related workloads where a task can be processed anywhere is of interest in our study because it is highly parallelizeable and independent of the other problems.

### 3.2.4 *Parallel Machine Scheduling*

Scheduling for parallel machines is more commonly used with High Performance Computing of various types: (a) cluster, (b) grid, and (c) cloud computing. In these types of HPC, nodes work in tandem to solve computationally difficult problems. Table 3.1 compares these three classes of HPC (Hussain et al. 2013).

As Fig. 3.1 shows (Dong and Akl 2006), scheduling algorithms used in HPC are characterized using five broad distinctions:

[7]See p.236

[8]See p.167

**Table 3.1** Comparison of cluster, grid, and cloud computing

| Feature | Cluster | Grid | Cloud |
|---|---|---|---|
| Size (# of nodes) | Small to medium (100's) | Large (1000's) | Small to large (100's–1000's) |
| Network type | Private LAN | Private LAN or WAN | Public WAN |
| Scheduling method | Centralized | Centralized or decentralized | Centralized or decentralized |
| Coupling | Tight | Loose or tight | Loose |
| SLA constraint | Strict | High | High |

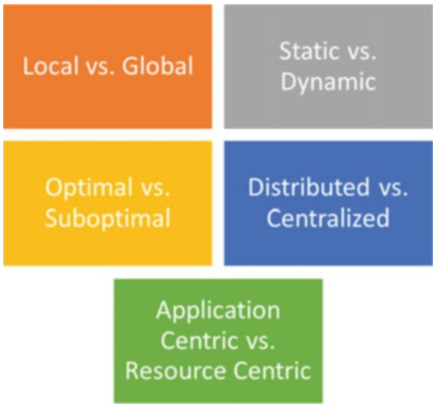

Local vs. Global
Static vs. Dynamic
Optimal vs. Suboptimal
Distributed vs. Centralized
Application Centric vs. Resource Centric

**Fig. 3.1** Classification of scheduling algorithms in HPC

– Global vs. Local. Global schedulers are unitary meta-schedulers that allocate resources across multiple administrative domains. In contrast, local schedulers manage smaller sets of resources and perform both sequencing and resource allocation.
– Static vs. Dynamic. Static schedulers allocate resources in advance and accept jobs only if sufficient resources are available. In contrast, dynamic schedulers queue jobs until sufficient resources become available.
– Optimal vs. Suboptimal. Because most scheduling problems faced by parallel systems cannot be computationally solved within a reasonable timeframe, most scheduling algorithms are suboptimal and heuristic.
– Centralized vs. Distributed. Centralized schedulers make every decision and have exclusive access to the resource pool. In contrast, distributed schedulers divide decisions among them and share a resource pool.
– Application Centric vs. Resource Centric. Application centric schedulers work to minimize the flow time for applications with specific load characteristics. In contrast, resource centric schedulers work to maximize resource utilization and increase flow time.

Dynamic scheduling is required when it is difficult to estimate the rate at which tasks will be submitted and/or the number of jobs that will request resources at a given time. This situation requires the scheduler to perform system-state estimation, which is computationally intensive because it requires the gathering and processing of complete information on the resources available and on every task in the system, including the number of CPUs, their memory, and their storage (Dong and Akl 2006). The difficulty of doing so further supports the FUD hypothesis, which states that the three goals of shared-computing systems—fairness, utilization, and dynamicity—cannot be optimized simultaneously. As a result of this hypothesis, many scheduling systems opt to increase resource utilization, even at the cost of increasing the time-in-system of some tasks (James 1999; Sedighi et al. 2017a).

## 3.3 Shared and High Performance Computing

Shared computing environments are platforms that allow many users to execute jobs simultaneously. Typically, the jobs that run in parallel on these systems could benefit from additional resources. High Performance Computing (HPC) environments are shared computing environments in which systems of interconnected computers increase the computational capacity available by providing parallelized execution paths across a pool of available resources. A resource pool is a set of computer nodes or servers ready to accept work at any time.

HPC environments must accommodate numerous and widely dispersed customers and usage profiles, each of which has priority and exhibits selfish behavior that may impact the other users in the system. Moreover, the users' tasks are malleable and have timescales measured in milliseconds (Feitelson et al. 1997). Thousands of tasks or requests are generated each second, and the number of users served at a given time can be large (Chervenak et al. 2000; Foster and Kesselman 2003).

Each user desires to gain access to as many resources as possible (Christodoulou et al. 2007). This selfish behavior is rational, and competition among users for resources is a zero-sum game. Little research has examined selfish behavior in shared computing environments, however, especially in interactive systems (Sedighi et al. 2014).

The type of problems under consideration are Massively Parallelizable Problems (MPPs) with dynamic and malleable tasks (Feitelson et al. 1997). The number of resources dedicated to such problems, which are composed of a number of tasks, may shrink or grow. When additional resources are allocated to the tasks pending for such a problem, that problem can be completed more quickly. For this reason, it is rational for the user to desire as many resources as possible. The problem arises when multiple users share this desire (Christodoulou et al. 2007; Sedighi et al. 2014): it creates a competition for resources.

Schedulers mediate access to resources, ensuring that no single user can dominate the environment. The primary goal of any scheduler is to "seem fair" (Kay and Lauder 1988). Because schedulers can be tasked with accommodating potentially large numbers of users and workloads, they must be able to divide resources fairly among users. However, some schedulers set aside resources for users (Bach 1986a, b) who are not able to claim them, thereby causing the environment to be underutilized.

Even though schedulers in HPC environments also ensure equifinality (Bertalanffy 1969)—i.e. that every user's job is finished—delays caused by unfair competition are undesirable (Ferraioli and Ventre 2009). Because the main objective of schedulers is to drain the queue of pending tasks as quickly as possible, schedulers pay very little attention to whom each task belongs. Although this model does reduce time-in-system and thereby ensures long-term fairness (Kay and Lauder 1988), temporal fairness suffers as a result. In essence, a user with only 10% of the pending tasks will be forced to wait considerably longer for resources than will users with more pending tasks.

# Chapter 4
# Fairshare Scheduling

Many scheduling systems use fair-share or proportional-fair-share algorithms (Kay and Lauder 1988). Fair-share schedulers were initially designed to manage the time allocations of processors in uniprocessor systems with workloads consisting of long-running, computer-bound processes (Kleban and Clearwater 2003). Each user was assigned a time slot on a machine (i.e. a mainframe), and in this time slot, the user's job was the highest priority. If there were any other jobs, they were stopped and restarted at a later time.

Kelban et al. argued that if the jobs that run on uniprocessor systems were not checkpoint-able (i.e. able to be stopped and restarted at a later time), fair-share scheduling would not be able to achieve real-time fairness. For this reason, fair-share scheduling policies were designed to achieve long-term fairness (i.e. steady-state fairness), for which immediate fairness was compromised (Kleban and Clearwater 2003). The scenarios discussed in the previous section, in which a faster response is desired for some tasks, pose a challenge to the current model. Scheduling policies like Lottery Scheduling (C. A. Waldspurger and Weihl 1994), Stride Scheduling (Carl A Waldspurger and Weihl 1995), Max-Min Fair-Share Scheduling, and Hierarchical Share Scheduling (Epema and de Jongh 1999) typically statically pre-assign a subset of the available resources. If these resources are not initially utilized, they are shared among the other tasks (Bui 2008).

Fair-share algorithms are simplified and tailored versions of lottery-based scheduling algorithm (C. A. Waldspurger and Weihl 1994). In lottery scheduling, lottery tickets are distributed in proportion to each user's number of tasks. For example, in a lottery-based system with 12 tickets, if *User 1* has 20 tasks pending and *User 2* has 10 tasks pending, *User 1* will be allocated 8 tickets and *User 2* will be allocated 4 tickets. In High Performance Computing environments, computational resources are "tickets," and schedulers assign resources in a ratio representing the current queue size of the pending tasks.

A scheduler monitors a queue of pending tasks and makes resource-allocation decisions dynamically. This dynamicity can produce dramatic shifts in how resources are allocated during a given scheduling cycle. Resources can be taken

A. Sedighi, M. Smith, *Fair Scheduling in High Performance Computing Environments*, https://doi.org/10.1007/978-3-030-14568-2_4

away from one user and assigned to another. Because the total number of resources is constant, one user's gain is another user's loss. Moreover, the number of resources available to a given user directly affects how fast their tasks are completed, making varying the resource allocation the primary method by which tasks get completed. The overall satisfaction or utility a user experiences results from the completion of their tasks.

Fair-share schedulers can fairly allocate Poisson-distributed workloads, but they cannot fairly allocate non-Poisson-distributed workloads, in which different users consume non-comparable amounts of resources (Kay and Lauder 1988). In the financial services industry, for instance, which uses HPC computing to assess the risk of stock portfolios (Tezuka et al. 2005), external events (such as market open and market close) can cause large numbers of tasks to be entered at nearly the same time. In such situations, the resulting workloads are not Poisson distributed and are not fairly allocated by fair-share schedulers.

## 4.1 Fairness

In social justice and welfare economics, the concept of fairness has been studied from a variety of perspectives, including Aristotle's equity principle (Shiner 1993), the theories of John Nash (Nash Jr. 1950) and John Rawls (Rawls 2009), and classical utilitarianism (Bertsimas et al. 2011):

- Utilitarianism: resources should be allocated to maximize the utility or satisfaction produced by the system.
- Aristotle's equity principle: resources should be allocated based on some pre-existing claims.
- Rawls's theory of justice: resources should be allocated to the users who are worst off to guarantee the highest level of increase in satisfaction and utility.
- Nash's bargaining theory: resources should be re-allocated if doing so will increase the utility of those to whom they are re-allocated to a greater degree than it will decrease the utility of those from whom they are taken (Young 1995).

One criticism of utilitarianism is that it would be unethical (Young 1995) to maximize the total utility generated by the system at the expense of the utilities of particular participants (Bertsimas et al. 2011). Despite this criticism, the utilitarian approach is adopted by the many scheduling systems that work to increase overall system utilization. Aristotle's principle is not relevant to our case because in our case, system resources are not pre-assigned. However, this principle is utilized in many scheduling systems (Bach 1986a, b; Henry 1984).

Rawls's and Nash's approaches to fairness can be used to model dynamic systems (Bertalanffy 1969). In any High Performance Computing Environment, some user is always the worst off. For this reason, the Rawlsian model of justice can be applied. In another scenario, a scheduler can determine the changes in utility for two users and use this information to make resource-management decisions. This relationship

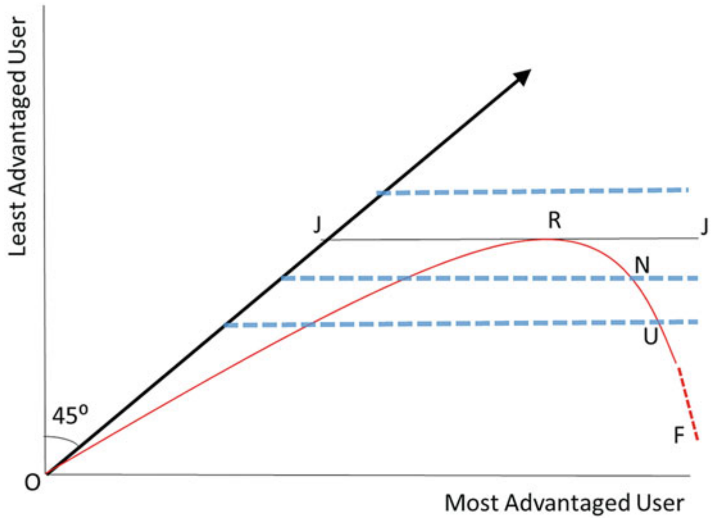


**Fig. 4.1** Fairness as it relates to different models (Sedighi et al. 2017b). Taken from (Rawls 2001)

is demonstrated in Fig. 4.1, where the Most Advantaged User (MAU) and Least Advantaged User (LAU) have equal utility and satisfaction at the 45°line. The JJ line (and all of the parallel, blue lines) represents all the MAU points that map to one fairness point on the LAU-axis, without lowering the satisfaction or utility to the LAU. Although it is difficult to achieve, the goal is to distribute resources as close to the 45°line as possible to achieve near equal utility for both users. The Rawlsian point (R) is the maximum point on the LAU-axis *and* the maximum point on the MAU-axis. As the distribution moves towards the Nash model (N) or the utilitarian model (U), the distribution benefits the MAU at a cost to the LAU. Any point to the left of the maximal R represents underutilization.

### *4.1.1 The Fairness of Fair-Share Scheduling*

The literature on scheduling describes two main types of fairness (Bui 2008) (Wierman 2011):

– Schedulers achieve "proportional fairness" when they schedule and dispatch smaller jobs first to reduce time-in-system.
– Schedulers achieve temporal fairness when, in cases in which two jobs have equal loads and runtimes, they schedule and dispatch first the job that arrived first—i.e. when they employ a first-come, first-served methodology.

Fair-share schedulers allocate resources to users in such a way that over a long period of time, each user gets their fair share of resources (Kay and Lauder 1988). As this statement implies, the primary goal of any scheduler is to seem fair to its users; even though "fairness" is not explicitly defined in literature, most agree that schedulers must "seem fair" (Kay and Lauder 1988) and that schedulers must divide resources "fairly" (Bui 2008). Very little research has been conducted into scheduling fairness, however (Wierman 2011) (Kleban and Clearwater 2003). Users may not agree on what constitutes "fair." For example, so that it could more quickly reduce processing queues and seem essentially fair, the Unix Scheduler (Bach 1986a, b) was designed to assign more resources to users with more pending processes. Similarly, the Proportional Share Scheduler (PSS) uses "required share" to statically pre-allocate a fraction of the resources available to each class of users; unused resources may be reassigned to other users via administrative interventions (Epema and de Jongh 1999). Every scheduling policy has focused mostly around the following optimization techniques (Sabin et al. 2004):

- Response Time: the time between when the job arrives and when it leaves the system.
- Wait Time: the time between when the job enters the system and when it is dispatched for execution.
- Slowdown: the ratio of response time to task size.
- Utilization: ratio of available resources in use.
- Capacity Loss: the ratio of available resources unutilized due to overhead, even when there are jobs pending in the system.

Although some schedulers have demonstrated improved performance in the aforementioned areas, some jobs still suffer from delays that are longer than expected (Sabin et al. 2004). In particular, jobs with low resource requirements and short processing times have been given higher priority in recent years (Bui 2008), and this has significantly increased the wait times experience by larger jobs. This policy embodies the utilitarian approach, on which some jobs are sacrificed to serve the "greater good."

## 4.2 Utility

A measure of the preference for reward over risk, utility is a critical concept in behavioral economics (Norstad 1999). The theory of utility stipulates that when given a choice among a number of options and outcomes, a rational person (or a rational "investor," to use economics terminology), will choose the option most likely to produce the best outcome for them. In order words, of two different options, the *preferred* option is the option expected to produce the highest utility value (Luce and Raiffa 2012). Utility theory has many applications, but it stands on its own as a means to explain preferences and desires (Luce and Raiffa 2012; Sugden 1991).

Some of the earliest accounts of the use of utility to explain and rationalize welfare on society are provided by Bentham (Bentham 1879), Edgeworth (Edgeworth 1879), Pareto (Pareto 1919), and Von Neumann, who used Game Theory

(Von Neumann and Morgenstern 2007) as the means to describe utility. Various views created a level of inconsistency, and this consistency is represented by the use of the concept of utility to explain users' happiness and to *predict* their actions by assuming that users share the desire for happiness. For example, the concept of utility can be used to explain investment decisions because it embodies the assumption that more wealth means more happiness (Norstad 1999).

There are two schools of thought regarding the measurement of utility:

– On Cardinal Utility, proposed by Jeremy Bentham (Bentham 1879), utility is conceived simply as the intensity of happiness multiplied by its duration. Bentham used this calculation to quantify social good. Edgeworth later transformed this measurement (Edgeworth 1879) into an integration over time, happiness and intensity to more precisely calculate societal happiness.
– On Ordinal Utility, proposed by Pareto (Pareto 1919), utility represents a choice between options. Pareto thought of utility as an index on which the differences between and preferences for two choices were important, not as a literal value given to the choices.

Von Neumann (Von Neumann and Morgenstern 2007) resurrected cardinal utility as a means to explain payoff functions in Game Theory. Von Neumann used utility as a measurement of the happiness produced by a given choice, where each choice was assigned a utility. However, Von Neumann still faced the problem of interpersonal comparability (Von Neumann and Morgenstern 2007), which states that the magnitude of happiness or satisfaction perceived by one user may not be comparable to that of another user. The utilitarian approach to fairness fails as a result of this problem (Bertsimas et al. 2011): it assumes that the long-term fairness of a system can satisfy all of its users in the short term.

### 4.2.1 Total Utility vs. Marginal Utility

If total utility is the ocean, marginal utility is the waves. If we want to know the volume of the ocean, measuring the waves does us no good (Georgescu-Roegen 1968). Measuring total happiness or goodness is very difficult because the problem of interpersonal comparability makes precise calculations virtually impossible. However, measuring marginal utility in the case in which we are measuring an additional "dose" of good is far simpler (Read 2004).

The second classification of utility distinguishes "experienced utility" from "decision utility" (Bentham 1879). The former represents the utility actually produced by a decision or action, while the latter represents the happiness that could possibly result from a decision yet to be made. In our work, we calculate marginal experienced utility (Layard 2003) based on a multi-criteria scheduler whose goal is to optimize a task's time-in-system.

## 4.3 FUD: Fairness-Utilization-Dynamicity

FUD (Sedighi et al. 2017b) states that a scheduler's three primary parameters—fairness, utilization, and dynamicity—cannot be optimized simultaneously: one of these three parameters must suffer for the sake of the other two.

- **F**airness: As described in Sect. 4.1, fairness deals with the mechanism by which resources are distributed. Most schedulers assume a utilitarian view of fairness (Sedighi et al. 2017a), but we assume a Rawlsian view of fairness.
- **U**tilization: utilization deals with overall system utilization. If a task needs to be restarted to be resubmitted, the wasted calculation time increases the underutilization of the system.
- **D**ynamicity: the degree to which a scheduler alters its decisions based on information received from the environment. The two extremes in this parameter are static scheduling and dynamic scheduling. In static scheduling, decisions are made ahead of time based on resource availability and job runtime. In dynamic scheduling, decisions are made based on received information in the form of events or telemetries.

The more events that are received from the environment and the higher the rate at which these events are received, the more time consuming and process intensive it is for a scheduler to decode the incoming events and to choose subsequent scheduling steps. A scheduler may choose to ignore events, but doing so could mean reaching less-informed decisions that could reduce utilization or fairness.

We determined that scheduling systems tend to optimize for dynamicity and utilization, reducing fairness in the process (Sedighi et al. 2017a).

# Chapter 5
# Multi-Criteria Scheduling: A Mathematical Model

## 5.1 Scope and Purpose

This section details a mathematical model that further explains the multi-criteria scheduling mechanism outlined in the main thesis. The traditional scheduling systems covered in Chap. 3 typically use two primary variables: load and priority. To avoid the limitations faced by such schedulers, the scheduler presented in our work uses as a third independent variable—seniority—to determine the order in which tasks are scheduled for execution. Unlike load and priority, seniority is system-calculated and changes over time. This aspect of this tertiary variable is also not modeled in scheduling systems.

This section models the relationships between these three variables and identifies the relationships they share with other variables in the literature, especially in the queuing system.

Add a figure to illustrate the procedure for a task to enter the execution by going through the queue and the scheduler. The main research is the scheduler by feedbacks form the queue and the execution units.

## 5.2 Scheduling Parameters in 2-Dimensional Space

Traditional scheduling systems use 2-dimensional space to model a given task (T), as shown in Fig. 5.1.

L and P are defined in the following sections.

A. Sedighi, M. Smith, *Fair Scheduling in High Performance Computing Environments*, https://doi.org/10.1007/978-3-030-14568-2_5

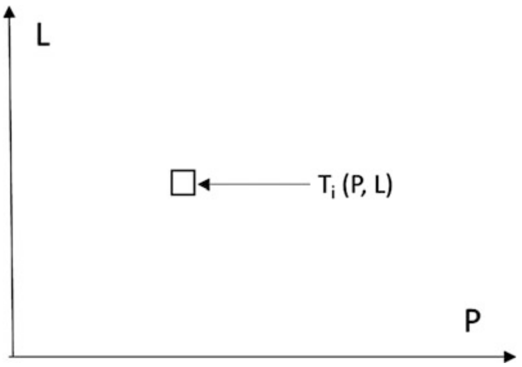


**Fig. 5.1** Scheduling parameters in 2-Dimensional space

### *5.2.1 Load Requirement of a Task (L)*

Traditionally, the load requirement of task was measured in CPU cycles per second. Because of recent advances in hardware architecture, however, L should be measured using the more-common floating-point operations per second (FLOPS). The load requirement of a given task can be given or deduced, and it does not change over the lifetime of the task. However, the load requirements of the tasks that constitute a given job may differ. For this reason, load is usually indexed to a particular task ($L_i$).

Every task has basic load and system requirements, and the scheduler needs to know these. A task's load and the computational capability of the system jointly determine how long the task takes to complete. In our work, task load was assumed to be static and the same for every task in a given job. This positive value is supplied by the user or deduced by the system and will not change once the task has entered the system.

$$L_i = constant \subset \mathbb{R}^+ \tag{5.1}$$

### *5.2.2 Priority of a Task (P)*

A task's priority is static and generally user-defined, and it determines the importance of the task. It varies from task to task and may change. Because P is defined by the user, P may not represent the priority of that task holistically across all the users and all tasks in the system. This value can be elevated by users' selfish behavior (Angel et al. 2006). This value may also be inaccurate because one user's view of priority may differ from those of others.

Because priority is defined by the user, it can be skewed. While every user believes that their tasks should be assigned the highest priority, the system may alter tasks' priority values in accordance with other criteria. In our work, we assume that the priority of each task is a constant and positive across the lifetime of the task.

$$P_i = constant \subset \mathbb{R}^+ \tag{5.2}$$

Task ($T_i$) is expressed in a column vector consisting of

$$T_i = \begin{pmatrix} P_i \\ L_i \end{pmatrix} \tag{5.3}$$

Scheduling algorithms use load and priority to decide which task to schedule next. As a result, the time required for a given task to leave the queue ($t_i$) is a function of Load (L) and Priority (P).

$$t_i = \mathcal{F}(\mathrm{L_i},\ P_i) \tag{5.4}$$

### 5.2.3 Auxiliary Parameters

Other system parameters that are affected by Load and Priority:

- The incoming task rate ($\lambda_j$) is the number of tasks that enter the scheduler per unit of time for a given user j.
- The outgoing task rate ($\mu_j$) is the number of tasks that leave the scheduler for execution per unit of time for a given user j.

A task is a unit of request which enters the system by a user. The states that tasks occupy are as follows:

- Tasks are Submitted and Queued when they are waiting in the queue to be scheduled.
- Tasks are Scheduled for Execution when they have left the scheduler and are running on the target system.
- Tasks are Completed when they have exited the target system.

## 5.3 Seniority of a Task

The scheduler developed in our work uses Load, Priority, and an additional primary parameter called Seniority. As a result, Eq. 5.5 is modified as follows:

$$T_i = \begin{pmatrix} S(t)_i \\ P_i \\ L_i \end{pmatrix} \tag{5.5}$$

Seniority, $S(t)$, is a dynamic, time-dependent variable that determines the duration that task has been in the system relative to the current state of the system, and relative to the other users currently in the system. A task's seniority changes with changes in the flux of the system, which will be described shortly.

A task's seniority indicates its "age" in the scheduler, though seniority changes as the result of other changes in the system. Seniority is measured in $\frac{1}{sec}$ and can, when combined with load, determine the temporal computational power requirement (FLOPS) of the task. This notion only works with malleable tasks, for which additional computational resources lower the completion time. For example, an increase in seniority from 1 to 2 would double the instantaneous FLOP required to meet the completion-time-fairness requirement of the task.

$$S_i(P, L; t) \subset \mathbb{R}^+ \tag{5.6}$$

## 5.4 Modeling Tasks in 3-Dimensional Space

A user j's task i ($T_{ij}$) represents a point in the 3-dimensional scheduler space, and a given job is a collection of such points in that space. Since task values can only be positive, a given task ($T$) can be assigned coordinates in a 3-dimensional semi-vector space (Janyška et al. 2007) over $\mathbb{R}^+$.

$$T \subseteq \mathbb{R}^{3+} \tag{5.7}$$

We can use Task Priority ($\boldsymbol{P}$), Task Load ($\boldsymbol{L}$), and Task Seniority ($\boldsymbol{S}(t)$) to represent tasks in a 3D coordinate system (Fig. 5.2).[1]

If every task has the same priority and load requirements, then the queue of pending tasks will be represented as a line parallel to the S-axis. If, however, the tasks have different load requirements and a different priority levels, a different envelope takes shape. This is the execution envelope for all of the tasks pending in the system. If the priority of a given task is directly related to its load requirements (5.8), the execution envelope is represented as a square perpendicular to the S axis.

$$P_i \Leftrightarrow L_i \tag{5.8}$$

The life-line of a task can thus be represented as a vector parallel to the S-axis in the positive 3-dimensional space ($\mathbb{R}^{3+}$). If the system permits the priority and load requirement of a given task to be changed after the task has entered the system, the vector representing the task could become any positive vector in 3-dimensional space. In our work, we assume that tasks' priorities and loads are constants.

[1]Although it is shown as a cube, it should be considered a single point in the 3-dimentional space.

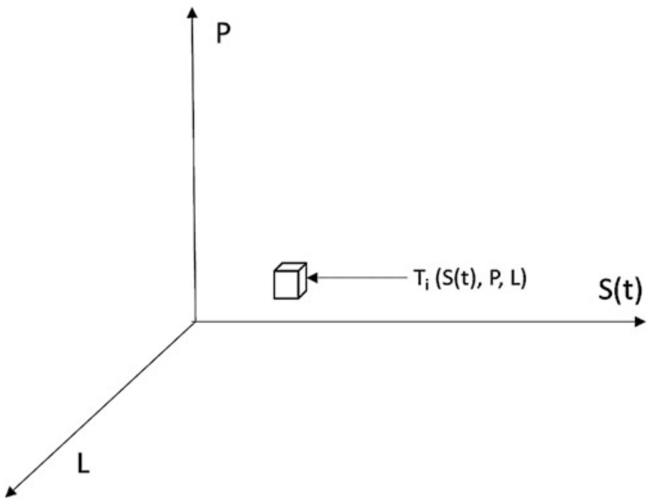


**Fig. 5.2** A single task modeled in 3-Dimensional space

**Fig. 5.3** Depiction of a task from queuing to being scheduled for execution

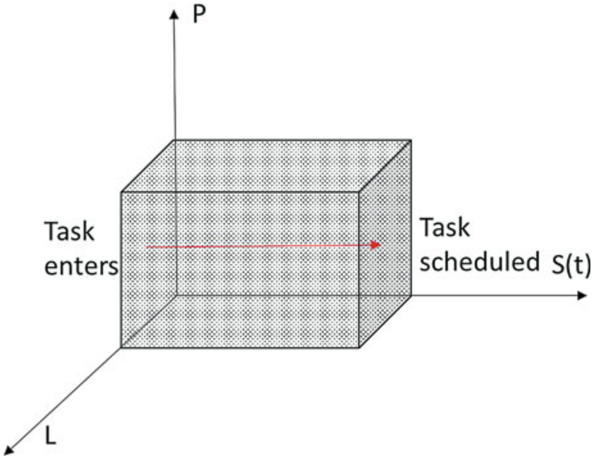


Figure 5.3 shows a single task among a collection of tasks encased in an execution envelope as per the aforementioned assumptions.

*Lemma 2.1* Tasks in the 3-dimensional scheduler space can be mapped into the classical, 2-dimensional scheduler space. Tasks in $\mathbb{R}^{3+}$ differ from tasks in the traditional $\mathbb{R}^{2+}$ in the following ways:

- Load (L) represents the computational power required per unit of time (i.e. seconds).
- Because Seniority (S) is missing, the 3D space collapses into a 2D space.
- From the above, it can be deduced that $f: T(\mathbb{R}^3) \overset{S=0}{\rightarrow} T(\mathbb{R}^2)$.

The prioritized and seniority-calibrated load requested by the pending tasks ($W(t)$) can be defined as the volume encapsulated by these three parameters.

$$W(t) = F(\boldsymbol{S}(t), \boldsymbol{P}, \boldsymbol{L}) \tag{5.9}$$

$$W(t) = \sum_{i=0}^{n} \int_{t1}^{t2} (\boldsymbol{P}_i \times \boldsymbol{L}_i) \cdot \boldsymbol{S}_i(t) dt \tag{5.10}$$

$W(t)$ is the total computational power required by the total number of tasks (represented by $n$), and $t_1$ and $t_2$ demarcate the duration under consideration.

As previously mentioned, seniority is a task property that is affected by other system behaviors. For this reason, seniority is unaffected if no tasks enter or leave the system.

$$\lambda_j = \mu_j = 0 \Rightarrow \Delta \boldsymbol{S}_i = 0 \forall i \tag{5.11}$$

It can be deduced from 5.12 that $\Delta \boldsymbol{S}_i > 0$ *if* tasks enter the system, even if none leave. Furthermore, tasks enter the system on the (P, L) plane with density distributions $J_j(S(t), P, L)$, which are related to the incoming and outgoing task densities as follows:

$$\lambda_j = \left. \frac{dJ_j(S(t), P, L)}{dt} \right|_{S=0} \tag{5.12}$$

$$\mu_j = \left. \frac{dJ_j(S(t), P, L)}{dt} \right|_{S=k} \tag{5.13}$$

$\lambda_j$ is the rate at which incoming tasks enter the system, and $\mu_j$ represents the rate at which tasks are scheduled for execution. Load density ($J_j$) represents the distribution of incoming tasks for user $j$ relative to load (L) and priority (P). User $j$ may submit any number of tasks at time = t, and each can have a different priority and load. The density determines the types of tasks entering the system in a specific period of time.

## 5.5 Determining Seniority and Fairness Factor

Assuming that no tasks are removed from the system, we can use the divergence theorem to determine task conservation in the enclosed $(S(t), P, L)$ envelope (5.14). Task conservation determines the dispensation rate of the tasks in the execution envelope. A small divergence causes the seniority of a given user's tasks to increase. Equation 5.18 determines the pending tasks as a function of tasks awaiting execution minus tasks entering the system as a function of time.

$$\int_{t_1}^{t_2} \iiint_V \left(div \cdot \vec{J}\right) dV dt = \int_{t_1}^{t_2} \oiint_{(P.L)} \vec{J} \cdot \hat{n}\, dA\, dt \tag{5.14}$$

Where:

$$div \cdot \vec{J} = \frac{\partial J_S}{\partial S} + \frac{\partial J_P}{\partial P} + \frac{\partial J_L}{\partial L} \tag{5.15}$$

With:

$$\frac{\partial J_P}{\partial P} = 0 \tag{5.16}$$

$$\frac{\partial J_L}{\partial L} = 0 \tag{5.17}$$

and

$$\oiint_{(P,L)} J \cdot \hat{n}\, dA = \oiint_{(P,L)} J_S(S = k) dL dP - \oiint_{(P,L)} J_S(S = 0) dL dP \tag{5.18}$$

The task conservation is thus:

$$\int_{t_1}^{t_2} \iiint_v \frac{\partial J_S}{\partial S} dV\, dt = \int_{t_1}^{t_2} \oiint_{(P,L)} J_S(S = k) dL\, dP\, dt - \int_{t_1}^{t_2} \oiint_{(P,L)} J_S(S = 0) dL\, dP\, dt \tag{5.19}$$

*Lemma 2.2* $div.J = \frac{\partial J_S}{\partial S}$ because of the assumption made in Eqs. 5.16 and 5.17: that load density does not change along the (P, L) axis.

Tasks enter the system and are distributed along the S-axis $\left(\frac{\partial J}{\partial S}\right)$. As a result, the left-hand side of the equation represents the total number of tasks pending execution:

$$\xi(t) = \int_{t_1}^{t_2} \iiint_V \frac{\partial J_S}{\partial S} dV dt \tag{5.20}$$

We stipulate that a movement in the +S direction is related directly to (P, L) as follows:

$$\frac{\Delta S_i}{\Delta t} = \alpha_i(t)(P_i\, L_i) \tag{5.21}$$

With total movement in the +S direction:

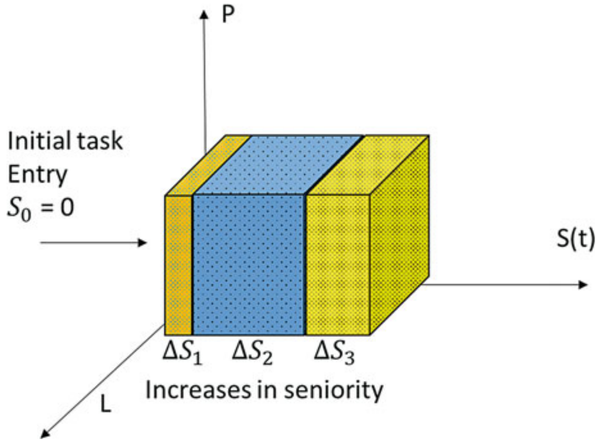


**Fig. 5.4** Depiction of additive seniority values

$$\mathrm{S}_i(\mathrm{t}) = \int_{t_1}^{t_2} \alpha_i(t)(P_i\, L_i)dt \tag{5.22}$$

This is better depicted in Fig. 5.4. The product of the load and the priority determines the directional movement of the seniority. $\alpha$ is the fairness factor or the fairness multiplier, which allows us to control the task diffusion ($T_i$) externally and thereby enforce a level of fairness across the system. $\alpha$ is calculated by taking into account the rate of pending tasks for a user relative to all the users:

$$\alpha_j^{-1}(t) = \frac{\xi_j(t)}{\sum_j \xi_j(t)} \tag{5.23}$$

The incoming flux of tasks pushes the tasks already in the system alongside the S-axis. The flux eventually pushes the task against the scheduling edge (the right-hand side of the execution envelope), indicating that it will be scheduled during the next scheduling cycle. The scheduler picks the tasks that have surfaced (Fig. 5.5) and pushes against the incoming tasks pending engagement.

The goal of the scheduler is to get to $S \Rightarrow 0$ as quickly as possible, and to do so, it needs to be able to schedule tasks at a rate equal to the rate at which tasks are moving in the +S direction:

- If $J_s(S = 0, j) = J_s(S = k, j)$, the pending task $\xi(t) = 0$, and $\alpha = 0$.
- If $\lambda_j > \mu_j$, the pending task $\xi(t) > 0$, and $\alpha > 0$.

We now need to calculate the total number of tasks pending at time $t$ for $\mathrm{S} = \mathrm{S}_k$. We can calculate the position of a single task at time t by summing X over the entire time the task has been traveling through the execution envelope.

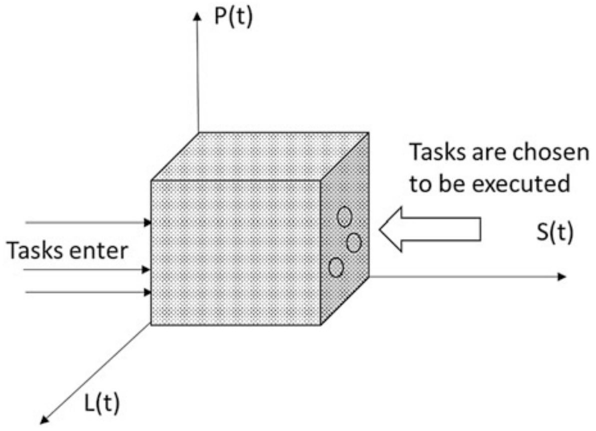


**Fig. 5.5** Tasks chosen for execution

## 5.6 Summary

Because our work introduced a scheduler that uses an additional variable, the traditional, 2-dimensional model featuring only load and priority had to be modified. The 3-dimensional model developed uses seniority to determine the status of a task in the queue before it is executed. This model, although crude, can be used as the foundation for a more comprehensive, unified theory that ties scheduling to the FUD conjecture detailed in Chap. 4.

# Chapter 6
Simulation and Methodology

As outlined in 1.3, our work assesses the performance characteristics of a multi-criteria scheduler that uses seniority as well as priority and load is to make decisions. It does so by using various simulation models to test the scheduling algorithm. Later in this book, the simulator used to conduct these simulations (dSim) is introduced.

Previous chapter described a shortcoming of fair-share scheduling techniques: they achieve long-term fairness at the expense of short-term fairness. Evidence of this includes cases in which system dynamics were altered by small and frequent requests. Further evidence is offered by Sedighi et al. (2014). Current scheduling techniques consider priority and load requirement in making scheduling decisions (Baker and Trietsch 2013), but do not employ a primary variable indicating the time that a task has spent in the system. Tasks are queued—or placed in multiple queues if they have different priority levels—and they are processed in the order in which they arrive. Queues are one-dimensional constructs in which load requirement is the primary independent variable. Priority queues are two-dimensional constructs in which priority is the second independent variable.

Our work introduces an additional independent variable that tracks the time each task has spent in the system. This variable is called "Seniority" (S); it is additive and increases in value as one would expect: a net-positive increase in seniority is calculated for each task after every scheduling cycle, and this new value is used to reorder the standings of the tasks prior to the next scheduling cycle. The task with the highest seniority is scheduled first for a given duple of priority and load set of tasks.

The rest of this chapter describes how seniority is calculated based on the mathematical model introduced in Chap. 5. The experiments conducted to assess the performance characteristics of the scheduler, which uses three independent variables: priority, load, and seniority will be introduced in Chap. 8.

A. Sedighi, M. Smith, *Fair Scheduling in High Performance Computing Environments*, https://doi.org/10.1007/978-3-030-14568-2_6

## 6.1 Calculating Seniority

Seniority (S), is a dynamic, time-dependent value calculated in part by the time that the task has been in the system relative to the current state of the system and to the other users in the system—i.e. the “time-in-system” (Baker and Trietsch 2013). The weight of seniority is determined by the fairness factor ($\alpha$). Seniority is an additive value: every time it is calculated, it is added to the seniority value that the task already holds. In essence, the longer a task has been pending (i.e. the larger is its time-in-system), the higher its seniority is. This relationship allows seniority to serve as a surrogate for time-in-system.

Change in seniority, S, is a function of priority and load,

$$\frac{dS}{dt} = \Delta S(t) = \alpha^* P^* L \tag{6.1a}$$

while total seniority depends on time-in-system ($t$):

$$S_{Total} = \sum_{t=0}^{t=current} \Delta S(t) \tag{6.1b}$$

where P represents the priority of the task, and L represents the load the task will put on the system. Both values are entered by the user when the task is submitted and (for our purposes) both remain constant while the task is in the system. Both values range from 0 to 1: 0 represents the lowest priority and the lowest computational load, and 1 represents the highest priority and the highest computational load. Equation 6.1b illustrates how a task’s seniority depends on the time that the task has been in the system. Seniority is calculated in discrete time intervals, but it is additive and continues to increase across the lifetime of the task.

The fairness factor, $\alpha$, is calculated by taking into account the rate of pending tasks for a user relative to all of the users:

$$\alpha_u^{-1}(t) = \frac{\xi(t)}{\sum_u \xi(t)} \qquad \#\,\text{Rawlsian} \tag{6.2}$$

Equation (5.2) implements Rawls’s philosophy of fairness, with $\xi(t)$ representing the pending tasks for a given user at time t and $\sum\limits_u \xi(t)$ representing the pending tasks for all of the users in the system. This equation causes the user with the least number of pending tasks to be moved to the front of the queue. This method resembles the shortest processing time (SPT) scheduling algorithm (Baker and Trietsch 2013), which has been shown to have the lowest makespan when scheduling jobs. In essence, the fairness factor makes an implicit assumption about the future task count of each user, and this assumption is adjusted if it is proven wrong in subsequent scheduling cycles.

For example, if User 1 (u1) has 10 pending tasks and User 2 (u2) has 20 pending tasks, then:

$$\alpha_{u1}^{-1}(t) = \frac{10}{30} = \frac{1}{3} \tag{6.3}$$

$$\alpha_{u1}(t) = 3 \tag{6.4}$$

meaning that for a given task, the fairness multiplier is 3 (Eq. 6.4). User 1's tasks move up 3 spots in the queue, but this does not mean that all of u2's tasks fall behind. As a result of the same calculation, user 2's tasks also move up:

$$\alpha_{u2}(t) = \left(\frac{20}{30}\right)^{-1} = \frac{3}{2} \tag{6.5}$$

In Nash's comparison model, the fairness factor is easier to calculate but more iterative:

$$U_u(t) = \frac{1}{\xi(t)} \qquad \# \text{Nash} \tag{6.6}$$

For $u = 1..n$, and:

$$a_{u1,u2}(t) = \frac{U_{u1}(t)}{U_{u2}(t)} \tag{6.7}$$

with $\xi(t)$ representing the number of tasks pending for a given user at time t and $U_u(t)$ representing the utility or satisfaction gained by a given user when an additional one of their tasks is executed. The fairness factor, $\alpha_{u1,\ u2}(t)$, relates u1 over u2: the fairness factor is the ratio of the utilities of two users competing for a resource.

In a Nash-based fairness-factor calculation, the utilities of all of the users are calculated. The fairness factor compares the utilities of two users. If user 1 has 10 pending tasks, when a single task is finished for user 1, user 1 gains a percentage change in utility of 1/10, 0.1, or 10%. Similarly, if user 2 has 30 pending tasks, their percentage gain in utility per task completed is 1/30, 0.03, or 3%.

$$\alpha_{u1,u2}(t) = \frac{0.1}{0.03} = 3.33 \tag{6.8}$$

$$\alpha_{u2,u1}(t) = \frac{0.03}{0.1} = 0.3 \tag{6.9}$$

In a direct competition between two users, Nash's model gives more precedence to the user with the lower number of pending tasks. Equation 6.10 shows the matrix for a 3-user system, and 6.11 shows a matrix for the general case:

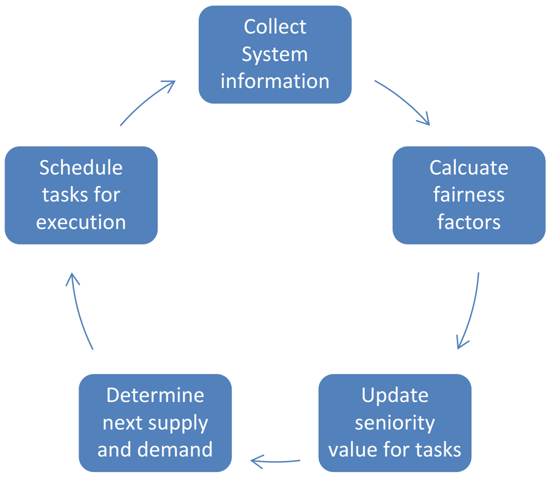

Collect System information
Calcuate fairness factors
Update seniority value for tasks
Determine next supply and demand
Schedule tasks for execution

**Fig. 6.1** The scheduler's process flow

$$\begin{bmatrix} n/a & \alpha_{u1,u2} & \alpha_{u1,u3} \\ \alpha_{u2,u1} & n/a & \alpha_{u2,u3} \\ \alpha_{u3,u1} & \alpha_{u3,u2} & n/a \end{bmatrix} \tag{6.10}$$

$$\begin{bmatrix} n/a & \cdots & \alpha_{u1,u_n} \\ \vdots & \ddots & \vdots \\ \alpha_{u_n,u1} & \cdots & n/a \end{bmatrix} \tag{6.11}$$

Nash's model is more computationally intensive than other methods of calculating utility, however, as it requires a comparison between all users. The main reason to choose one over the other is computational efficacy.

The scheduler is itself a new operating definition for resource management, as is shown in Fig. 6.1. The scheduler achieves fairness because if it overshoots, incorrectly bumping up a task, the next task to be processed for that user will stay in place or move very slightly towards the front of the queue.

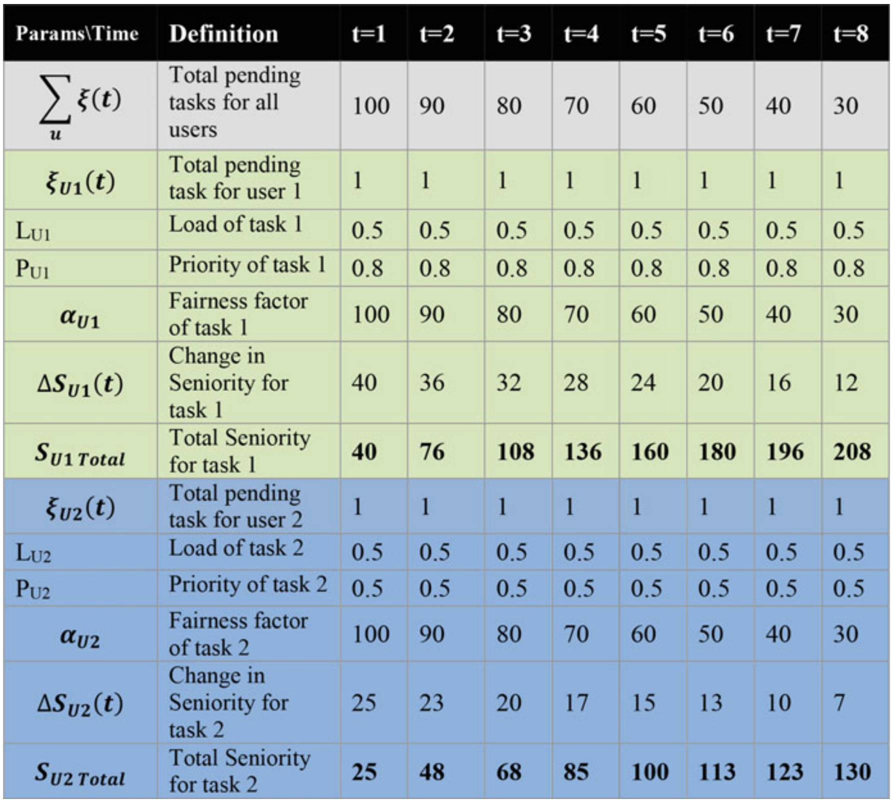
### 6.1.1 *Example of Calculating Seniority*

Consider the following scenarios. In the first scenario, two users of a large system each submit one task. Task 1 is submitted by user 1, and task 2 is submitted by user 2 (Table 6.1). While user 1 and user 2 each submit only a single task, the system contains other tasks submitted by other users, which are represented by $\sum_u \xi(t)$ (Table 6.2). In the second scenario, user 1` has two pending tasks, while the other conditions are the same as in the first scenario (Tables 6.1 and 6.3).

**Table 6.1** Parameters for calculating seniority in a 2-user example

| | User 1 | User 2 | User 1 |
|---|---|---|---|
| Number of tasks | 1 | 1 | 2 |
| Load | {0.5} | {0.5} | {0.5} |
| Priority | {0.8} | {0.5} | {0.8} |
| Fairness factor (estimated) | Rawlsian | Rawlsian | Rawlsian |

**Table 6.2** Calculating seniority in a 2-user example

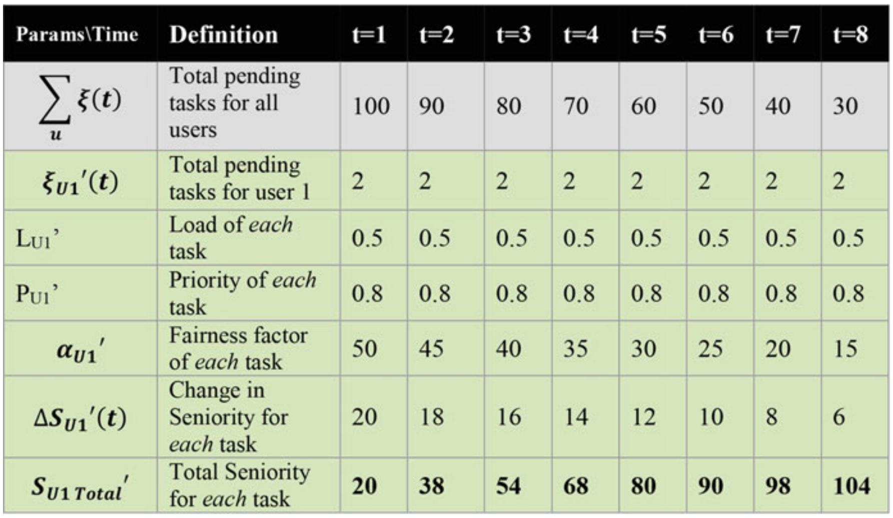


| Params\Time | Definition | t=1 | t=2 | t=3 | t=4 | t=5 | t=6 | t=7 | t=8 |
|---|---|---|---|---|---|---|---|---|---|
| $\sum_u \xi(t)$ | Total pending tasks for all users | 100 | 90 | 80 | 70 | 60 | 50 | 40 | 30 |
| $\xi_{U1}(t)$ | Total pending task for user 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| $L_{U1}$ | Load of task 1 | 0.5 | 0.5 | 0.5 | 0.5 | 0.5 | 0.5 | 0.5 | 0.5 |
| $P_{U1}$ | Priority of task 1 | 0.8 | 0.8 | 0.8 | 0.8 | 0.8 | 0.8 | 0.8 | 0.8 |
| $\alpha_{U1}$ | Fairness factor of task 1 | 100 | 90 | 80 | 70 | 60 | 50 | 40 | 30 |
| $\Delta S_{U1}(t)$ | Change in Seniority for task 1 | 40 | 36 | 32 | 28 | 24 | 20 | 16 | 12 |
| $S_{U1\,Total}$ | Total Seniority for task 1 | **40** | **76** | **108** | **136** | **160** | **180** | **196** | **208** |
| $\xi_{U2}(t)$ | Total pending task for user 2 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| $L_{U2}$ | Load of task 2 | 0.5 | 0.5 | 0.5 | 0.5 | 0.5 | 0.5 | 0.5 | 0.5 |
| $P_{U2}$ | Priority of task 2 | 0.5 | 0.5 | 0.5 | 0.5 | 0.5 | 0.5 | 0.5 | 0.5 |
| $\alpha_{U2}$ | Fairness factor of task 2 | 100 | 90 | 80 | 70 | 60 | 50 | 40 | 30 |
| $\Delta S_{U2}(t)$ | Change in Seniority for task 2 | 25 | 23 | 20 | 17 | 15 | 13 | 10 | 7 |
| $S_{U2\,Total}$ | Total Seniority for task 2 | **25** | **48** | **68** | **85** | **100** | **113** | **123** | **130** |

The change in seniority is calculated using Eq. (5.1), and the total accumulated seniority is calculated using Eq. (5.1). In both scenarios, the fairness factor $\alpha$ is designed to implement the Rawlsian definition of fairness, as is suggested by Eq. (5.2).

The total number of tasks pending system wide starts at 100 and decreases every clock cycle. As the time spent in the queue (i.e. the time-in-system) of a task

**Table 6.3** Recalculating seniority with an additional task pending for user 1



| Params\Time | Definition | t=1 | t=2 | t=3 | t=4 | t=5 | t=6 | t=7 | t=8 |
|---|---|---|---|---|---|---|---|---|---|
| $\sum_u \xi(t)$ | Total pending tasks for all users | 100 | 90 | 80 | 70 | 60 | 50 | 40 | 30 |
| $\xi_{U1}'(t)$ | Total pending tasks for user 1 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |
| $L_{U1}$’ | Load of *each* task | 0.5 | 0.5 | 0.5 | 0.5 | 0.5 | 0.5 | 0.5 | 0.5 |
| $P_{U1}$’ | Priority of *each* task | 0.8 | 0.8 | 0.8 | 0.8 | 0.8 | 0.8 | 0.8 | 0.8 |
| $\alpha_{U1}'$ | Fairness factor of *each* task | 50 | 45 | 40 | 35 | 30 | 25 | 20 | 15 |
| $\Delta S_{U1}'(t)$ | Change in Seniority for *each* task | 20 | 18 | 16 | 14 | 12 | 10 | 8 | 6 |
| $S_{U1\,Total}'$ | Total Seniority for *each* task | **20** | **38** | **54** | **68** | **80** | **90** | **98** | **104** |

increases, so does its seniority value. Because it has a higher priority value, user 1’s task increases in seniority at a higher rate than does user 2’s task.

In scenario 2—which differs from scenario 1 only in that user 1 has 2 pending tasks—the seniority value of user 1 drops by ½ ($S_{U1\ Total} = 104$) to compensate for the increased number of pending tasks, becoming smaller than the seniority value of user 2’s task ($S_{U2\ Total} = 130$). This result is shown in Table 6.3.

## 6.2 Performance Measures

As mentioned in Sect. 1.5, the primary objective of our work was to assess the performance characteristics of a multi-criteria scheduler that utilizes seniority as well as load and priority. Time-in-system was a primary measure in evaluating the scheduler’s performance in accommodating various sizes of tasks, classes of workloads, and numbers of users.

Another performance measure, expected utility, which is used to represent user satisfaction, was calculated using the Rawlsian method of fairness. Utility is calculated by comparing the number of completed tasks to the total number of tasks:

$$U_{u_T}(t) = \frac{\xi(t=0) - \xi(t)}{\xi(t=0)} \tag{6.12}$$

where $U_{u_T}(t)$ is the total utility (T) of the user (u) at time t. $\xi(t = 0)$ represents the total number of tasks pending at t = 0—i.e. the total tasks submitted by a given user—and $\xi(t)$ is the current number of pending tasks. A utility plot was used to validate our algorithm.

The algorithm makes scheduling decisions in time intervals, each of which is called a "bucket." Each bucket is filled with incoming tasks. The scheduler makes resource-allocation (i.e. scheduling) decisions for each bucket in the order in which it arrives. Each bucket is then fully processed. For each bucket, the following information is gathered:

– Bucket size
  - The bucket size represents the number of tasks each user submits per scheduling cycle. This value is used to calculate time-in-system for each task and utility of the user.
– Completion time
  - The completion time is used to calculate inter-bucket differences in utility and time-in-system.
– Time-in-System
  - As mentioned previously, a given job can be composed of many tasks that may enter the system at any time. The time-in-system of a given job is a meaningless value because if 2 tasks compose a job, one task may be submitted at the beginning of the day and the other may be submitted at the end of the day. Part of the analysis was devoted to determining the times-in-system of tasks from the various usage profiles we simulated.

In essence, the algorithm time-boxes the utility calculation to ensure that we are within operating parameters.

Bucket duration was also examined. Because each bucket had its own metric, a clear picture of utility emerges. In order to maintain fairness in statistical control, we monitored these values to make sure that the bucket completion time remained in an envelope acceptable to the operators, users, and business. If not, the fairness factor for the user needs to be revisited.

## 6.3 Experimental Simulation Methodology

The primary goals of the simulation were to validate the new scheduling algorithm and to determine its efficacy in handling various types of workloads. A number of simulators were available, including Optorsim (Bell et al. 2003), the Bricks Grid simulator (Takefusa et al. 2003), GridFlow (Cao et al. 2003), GridSim (Buyya and Murshed 2002), and Alea (Klusáček and Rudová 2010). The majority of these simulators were designed for job scheduling, however, and thus lacked the characteristics necessary to handle short-running tasks. In addition, we required a simulator that was extendable and able to accommodate dynamic sets of resources, job types, submission profiles, and task durations. We created dSim for this purpose.

# Chapter 7
# DSIM

dSim improves upon Alea 3.0 (Klusacek 11/17/2014; Klusáček and Rudová 2010) and GridSim (Albin et al. 2007; Buyya and Murshed 2002). GridSim is a discrete, event-based toolkit that uses the Java language (McGill 2008) (p.56–75). It allows components to be modelled and simulated in parallel environments, and it can be used to evaluate scheduling algorithms. It can also create different classes of heterogeneous resources and enables such resources to be managed using a user-defined scheduling algorithm. Even though GridSim allows environments to be managed via a graphical user interface, GridSim's framework was of greatest interest because it allowed us to customize our interactions with the environment. Alea expands upon the GridSim framework, enabling the evaluation of various scheduling techniques. dSim extends the framework created by Alea 3.0, enabling scenarios to be automated and simulated by varying the number of users, the types of job profiles, and the number of resources. Most importantly, it includes a simulator clock. dSim uses a (bucket) Placement Counter (PC) to keep all of the tasks synchronized in buckets. Each tick of the PC is a new bucket, in which events like task submissions, gatherings of results, and scheduling events are synchronized. The duration of a single PC tick was set at 1000 ms (1 s.).

We disabled Alea's graphical presentation of results and outputted our results to files containing the data required to measure the scheduler's performance. We analyzed this data externally using Input Analyzer, a stand-alone feature of Arena ("Rockwell Automation" 2016) that allows data to be visualized.

## 7.1 dSim Architecture and Simulation Model

Although dSim is based on Alea's framework (Klusacek 11/17/2014), it overcomes a number Alea's of limitations. For one, Alea reads job information from input files, which are time-consuming to generate. dSim includes a job-creation plug-in that automates this process, enabling the programmatic configuration of number of users, number of jobs, job frequency, size of payload, and basic information like user name, time of task submission, and task duration. StrategyMultiUserGeneric was the class created to simulate loads, and it has the following basic control functions:

```
//implementation omitted
//some of the auxiliary functions omitted
class StrategyMultiUserGeneric {
    public void setNumberOfSubmitters(int count);
    public String getNextUserID() throws Exception;
    public int getNextJobID();
    public void setWavelength(String lambda);
    public void setAmplitude(String load);
    public void setPhase(String phase);
    public void setTaskCount(String count);
    public void incrementProgramCounter();
}
```

To simulate the sharing of an environment by many users, we first needed to add the ability to control the number of users. We did this by calling `setNumberOfSubmitters(int count)`. Job profiles or submissions can be thought of as waves of input information. To control these profiles, we used wave semantics to describe and implement the job-loader class.

The ability to fine-tune job-submission profiles was achieved by calling `setTaskCount(String count)`, `setWavelength(String lambda)`, `setAmplitude(String load)`, and `setPhase(String phase)`. The passed strings take the form of colon-delimited numbers (not ratios). For example:

```
    :
(1) simulator.setNumberOfSubmitters(4);
(2) simulator.setTaskCount(1000:200:200:50);
(3) simulation.setWavelength("4:2:2:1");
(4) simulation.setAmplitude("10:10:20:100");
(5) simulation.setPhase("0:0:0:1");
    :
```

(1) sets the number of users to 4, and they show up in the output as: User1, User2, User3, and User4. (2) sets the total number of jobs that a given user will submit over the duration of the simulation. In the current example, the total number of tasks submitted by user 1 is 1000, the total number of tasks submitted by user 2 is 200, and so on. The simulation continues until all of the users have submitted their tasks and received their results. While some users may receive their results before others do, these users simple remain idle for the remainder of the simulation. (3) sets the wavelengths. User1 submits 10 jobs for every 4 PCs, User2 submits 10 jobs for every 2 PCs, and so on. The number of tasks sent to the scheduler per PC is set by setting the amplitude in (4). One deviation from wavelength semantics is wavelength

0, which indicates a constant load. (5) indicates the start time for each user, starting with PC 0. It should be noted that actual numbers are used, not ratios. If the amplitude is given as “8:6:4:2,” even though it looks the same from a ratio perspective, it is very different from a job-submission perspective. For a given string, for every 8 tasks that User1 submits, User2 submits 6, User3 submits 4, and User4 submits 2.

The API (Application Programming Interface) allows, for example, to simulate two users with loads resembling the following:

$$\mathrm{L}_1(\mathrm{x}, \text{Placement Counter}) = 1000, \sum_{PC=0}^{n} L_1(PC) = 500 \tag{7.1}$$

$$\mathrm{L}_2(\mathrm{x}, \mathrm{PC}) = \max\{\mathrm{L}_1(\mathrm{x}, \mathrm{PC}) \sin(\pi^* \mathrm{PC} + 3\pi/2), 0\}, \sum_{PC=0}^{n} L_2(PC) = 1000 \tag{7.2}$$

Here, PC = 1, . . . , n. $\mathrm{L}_1(\mathrm{x}, \mathrm{PC})$ is the base load of 1000 tasks/PC, and $\mathrm{L}_2(\mathrm{x}, \mathrm{PC})$ is 1000 tasks for every other PCs. $\mathrm{L}_1$ sends a total of 500 tasks over the course of the simulation, and $\mathrm{L}_2$ sends a total of 1000 tasks. The aforementioned load can be represented as follows (Fig. 7.1):

```
    :
simulator.setNumberOfSubmitters(2);
simulation.setTaskCount("500:10000");
simulation.setAmplitude("1000:1000");
simulation.setWavelength("0:2");
simulation.setPhase("0:1");
    :
```

The next step in the simulation is the scheduling of incoming tasks, which is conducted via the `NewFairShare` class. The `NewFairShare` class is the implementation of the fair-share algorithm used in the simulation. This class has three

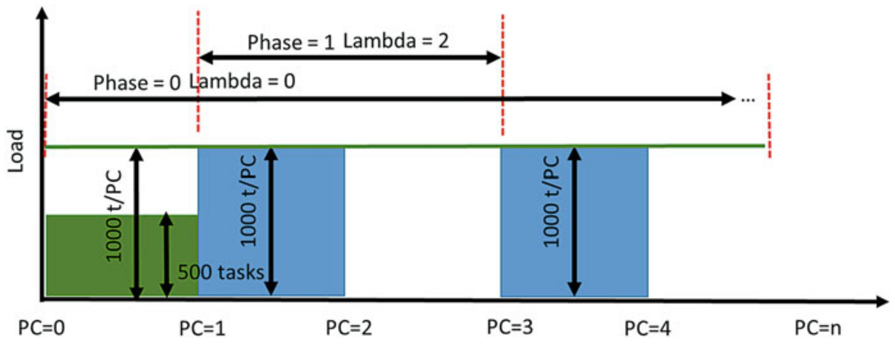


**Fig. 7.1** Illustration of the task-submission strategies of two users

public interfaces and one private interface, which is called after each new task added to the queue. Every user has an "incoming" queue, and this scheduling step employs the proportional fair-share algorithm outlined in C. A. Waldspurger and Weihl (1994). The step is triggered when the queue of tasks waiting to be dispatched—which is controlled by `setGangSize`—is empty. The task id of the next task to be dispatched and executed is returned by calling `selectTask`. The task duration can be changed, but we chose to hold it equal to one PC tick.

```
//implementation omitted
public class NewFairShare implements SchedulingPolicy {
    public void setGangSize(int size);
    private void calculateRatios(int total);
    public void addNewTask(GridletInfo gi);
    public int selectTask();
}
```

Each task comes with a submission time, which is entered by the user when they submit the task. Submission time and completion time are used to determine the time-in-system of a task. Since all of the tasks in the simulation require the same computation time (one PC tick), it is possible to measure and compare the delays experienced by two or more users over the course of the simulation.

## 7.2 dSim Configuration

dSim reads a configuration file (config.properties) that holds all of the parameters necessary to run a given simulation, including number of users, number of tasks per user, each user's submission profile and pattern, output file location, and debug file location. Each configuration takes the form of "name = value" and resembles the following:

```
### majority of parameters omitted
    :
number-of-submitters=2
task-count=500:10000
task-amplitude=1000:1000
task-wavelength=0:2
task-phase=0:1
    :
```

# Chapter 8
# Simulation Scenarios

Table 8.1 outlines the seventeen simulated scenarios run in our work. The goal of these simulations was to simulate the model proposed in Sect. 8.2. These simulations cover the four classes of job profiles discussed in Sect 1.2.

The first group of simulations covered the 2-user scenario, in which two users compete for access to system resources. The number of users in the simulation was then expanded to 6, 11, and 21. The total number of resources was held constant at 100 across the simulations (see Sect. 8.14 for a summary of assumptions) (Fig. 8.1).

## 8.1 Simulation Case 1–2 Users Base Case

This two-user, trivial case demonstrates the use of the simulator and its configuration. The total load (1000 tasks per user) is submitted at the beginning of the simulation.

$$L_1(x, PC) = \max\{1000 * L_0(x, PC)\cos(\pi * PC), 0\} \tag{8.1a}$$

$$L_2(x, PC) = \max\left\{1000 * L_0(x, PC)\sin\left(\pi * PC + \frac{3\pi}{2}\right), 0\right\} \tag{8.1b}$$

$$L_0(x, PC) = 1 \tag{8.2}$$

$$\sum_{PC=0}^{n} L_1(PC) = \sum_{PC=0}^{n} L_2(PC) = 1000 \tag{8.3}$$

A. Sedighi, M. Smith, *Fair Scheduling in High Performance Computing Environments*, https://doi.org/10.1007/978-3-030-14568-2_8

**Table 8.1** Simulation scenarios covered by this research

| Simulation number | Number of users | Job profile | Job profile description |
|---|---|---|---|
| 1 | 1 | $L_1(x, PC) = \max\{200^* L_0(x, PC)\cos(\pi^* PC), 0\}, \sum_{PC=0}^{n} L_1(PC) = 1000$ | 8.1 |
| | | $L_2(x, PC) = \max\{200^* L_0(x, PC)\sin(\pi^* PC + 3\pi/2), 0\}, \sum_{PC=0}^{n} L_1(PC) = 1000$ | |
| 2 | 2 | $L_1(x, PC) = \max\{1000^* L_0(x, PC)\cos(\pi^* PC), 0\}, \sum_{PC=0}^{n} L_1(PC) = 5000$ | 8.2 |
| | | $L_2(x, PC) = \max\{500^* L_0(x, PC)\sin(\pi^* PC + 3\pi/2), 0\}, \sum_{PC=0}^{n} L_2(PC) = 5000$ | |
| 3 | 2 | $L_1(x, PC) = 100^* L_0(x, PC), \sum_{PC=0}^{n} L_1(PC) = 5000$ | 8.3 |
| | | $L_2(x, PC) = \max\{200^* L_0(x, PC)\cos(\pi^* PC), 0\}, \sum_{(PC=0)}^{n} L_2(PC) = 5000$ | |
| 4 | 2 | $L_1(x, PC) = \max\{5000^* L_0(x, PC)\cos(\pi^* PC), 0\}, \sum_{PC=0}^{n} L_1(PC) = 5000$ | 8.4 |
| | | $L_2(x, PC) = \max\{50^* L_0(x, PC)\sin(\pi^* PC + 3\pi/2), 0\}, \sum_{PC=0}^{n} L_2(PC) = 50$ | |
| 5 | 2 | $L_1(x, PC) = \max\{5000^* L_0(x, PC)\cos(\pi^* PC), 0\}, \sum_{PC=0}^{n} L_1(PC) = 5000$ | 8.5 |
| | | $L_2(x, PC) = \max\{L_0(x, PC)\cos(\pi^* PC), 0\}, \sum_{PC=0}^{n} L_2(PC) = 10$ | |
| 6 | 6 | $L_1(x, PC) = \max\{200^* L_0(x, PC)\sin(\pi^* PC), 0\}, \sum_{PC=0}^{n} L_1(PC) = 1000 L_1(x, PC) = L_2(x, PC) = L_3(x, PC) = L_4(x, PC) = L_5(x, PC)$ | 8.6 |
| | | $L_6(x, PC) = \max\{500^* L_0(x, PC)\sin(\pi^* PC + 3\pi/2), 0\}, \sum_{PC=0}^{n} L_2(PC) = 5000$ | |

| | | | |
|---|---|---|---|
| 7 | 6 | $\mathrm{L}_1(\mathrm{x,PC}) = 20^*\ \mathrm{L}_0(\mathrm{x,PC}),\ \sum_{PC=0}^{n} L_1(PC) = 1000\ \mathrm{L}_1(\mathrm{x,PC}) = \mathrm{L}_2(\mathrm{x,PC}) = \mathrm{L}_3(\mathrm{x,PC}) = \mathrm{L}_4(\mathrm{x,PC}) = \mathrm{L}_5(\mathrm{x,PC})$ | 8.7 |
| | | $\mathrm{L}_6(\mathrm{x,PC}) = \max\{200^*\ \mathrm{L}_0(\mathrm{x,PC})\cos(\pi^*\ \mathrm{PC}), 0\},\ \sum_{PC=0}^{n} L_6(PC) = 5000$ | |
| 8 | 6 | $\mathrm{L}_1(\mathrm{x,PC}) = \max\{5000^*\ \mathrm{L}_0(\mathrm{x,PC})\cos(\pi^*\ \mathrm{PC}), 0\},\ \sum_{PC=0}^{n} L_1(PC) = 1000\mathrm{L}_1(\mathrm{x,PC}) = \mathrm{L}_2(\mathrm{x,PC}) = \mathrm{L}_3(\mathrm{x,PC}) = \mathrm{L}_4(\mathrm{x,PC}) = \mathrm{L}_5(\mathrm{x,PC})$ | 8.8 |
| | | $\mathrm{L}_6(\mathrm{x,PC}) = \max\{50^*\ \mathrm{L}_0(\mathrm{x,PC})\sin(\pi^*\ \mathrm{PC} + 3\pi/2), 0\},\ \sum_{PC=0}^{n} L_6(PC) = 50$ | |
| 9 | 6 | $\mathrm{L}_1(\mathrm{x,PC}) = \max\{1000^*\ \mathrm{L}_0(\mathrm{x,PC})\cos(\pi^*\ \mathrm{PC}), 0\},\ \sum_{PC=0}^{n} L_1(PC) = 1000\mathrm{L}_1(\mathrm{x,PC}) = \mathrm{L}_2(\mathrm{x,PC}) = \mathrm{L}_3(\mathrm{x,PC}) = \mathrm{L}_4(\mathrm{x,PC}) = \mathrm{L}_5(\mathrm{x,PC})$ | 8.9 |
| | | $\mathrm{L}_6(\mathrm{x,PC}) = \max\{\ \mathrm{L}_0(\mathrm{x,PC})\cos(\pi^*\ \mathrm{PC}), 0\},\ \sum_{PC=0}^{n} L_6(PC) = 10$ | |
| 10 | 11 | $\mathrm{L}_1(\mathrm{x,PC}) = \max\{100^*\ \mathrm{L}_0(\mathrm{x,PC})\cos(\pi^*\ \mathrm{PC}), 0\},\ \sum_{\mathrm{PC}=0}^{\mathrm{n}} \mathrm{L}_1(\mathrm{PC}) = 500\mathrm{L}_1(\mathrm{x,PC}) = \ldots = \mathrm{L}_{10}(\mathrm{x,PC})$ | 8.10 |
| | | $\mathrm{L}_{11}(\mathrm{x,PC}) = \max\{500^*\ \mathrm{L}_0(\mathrm{x,PC})\sin(\pi^*\ \mathrm{PC} + 3\pi/2), 0\},\ \sum_{\mathrm{PC}=0}^{\mathrm{n}} \mathrm{L}_{11}(\mathrm{PC}) = 5000$ | |
| 11 | 11 | $\mathrm{L}_1(\mathrm{x,PC}) = 10^*\ \mathrm{L}_0(\mathrm{x,PC}),\ \sum_{PC=0}^{n} L_1(PC) = 500\mathrm{L}_1(\mathrm{x,PC}) = \ldots = \mathrm{L}_{10}(\mathrm{x,PC})$ | 8.11 |
| | | $\mathrm{L}_{11}(\mathrm{x,PC}) = \max\{200^*\ \mathrm{L}_0(\mathrm{x,PC})\cos(\pi^*\ \mathrm{PC}), 0\},\ \sum_{PC=0}^{n} L_{11}(PC) = 5000$ | |
| 12 | 11 | $\mathrm{L}_1(\mathrm{x,PC}) = \max\{500^*\ \mathrm{L}_0(\mathrm{x,PC})\cos(\pi^*\ \mathrm{PC}), 0\},\ \sum_{PC=0}^{n} L_1(PC) = 500\mathrm{L}_1(\mathrm{x,PC}) = \ldots = \mathrm{L}_{10}(\mathrm{x,PC})$ | 8.12 |
| | | $\mathrm{L}_{11}(\mathrm{x,PC}) = \max\{50^*\ \mathrm{L}_0(\mathrm{x,PC})\sin(\pi^*\ \mathrm{PC} + 3\pi/2), 0\},\ \sum_{PC=0}^{n} L_{11}(PC) = 50$ | |

(continued)

**Table 8.1** (continued)

| Simulation number | Number of users | Job profile | Job profile description |
|---|---|---|---|
| 13 | 11 | $L_1(x, PC) = \max\{500^* L_0(x, PC)\cos(\pi^* PC), 0\}, \sum_{PC=0}^{n} L_1(PC) = 500L_1(x, PC) = \ldots = L_{10}(x, PC)$ | 8.13 |
| | | $L_{11}(x, PC) = \max\{L_0(x, PC)\cos(\pi^* PC), 0\}, \sum_{PC=0}^{n} L_{11}(PC) = 10$ | |
| 14 | 21 | $L_1(x, PC) = \max\{50^* L_0(x, PC)\cos(\pi^* PC), 0\}, \sum_{PC=0}^{n} L_1(PC) = 250L_1(x, PC) = \ldots = L_{20}(x, PC)$ | 8.10 |
| | | $L_{21}(x, PC) = \max\{500^* L_0(x, PC)\sin(\pi^* PC + 3\pi/2), 0\}, \sum_{PC=0}^{n} L_{21}(PC) = 5000$ | |
| 15 | 21 | $L_1(x, PC) = 10^* L_0(x, PC), \sum_{PC=0}^{n} L_1(PC) = 250 L_1(x, PC) = \ldots = L_{20}(x, PC)$ | 8.11 |
| | | $L_{21}(x, PC) = \max\{200^* L_0(x, PC)\cos(\pi^* PC), 0\}, \sum_{PC=0}^{n} L_{21}(PC) = 5000$ | |
| 16 | 21 | $L_1(x, PC) = \max\{250^* L_0(x, PC)\cos(\pi^* PC), 0\}, \sum_{PC=0}^{n} L_1(PC) = 250L_1(x, PC) = \ldots = L_{20}(x, PC)$ | 8.12 |
| | | $L_{21}(x, PC) = \max\{50^* L_0(x, PC)\sin(\pi^* PC + 3\pi/2), 0\}, \sum_{PC=0}^{n} L_{21}(PC) = 50$ | |
| 17 | 21 | $L_1(x, PC) = \max\{250^* L_0(x, PC)\cos(\pi^* PC), 0\}, \sum_{PC=0}^{n} L_1(PC) = 250L_1(x, PC) = \ldots = L_{20}(x, PC)$ | 8.13 |
| | | $L_{21}(x, PC) = \max\{L_0(x, PC)\sin(\pi^* PC), 0\}, \sum_{PC=0}^{n} L_{21}(PC) = 10$ | |

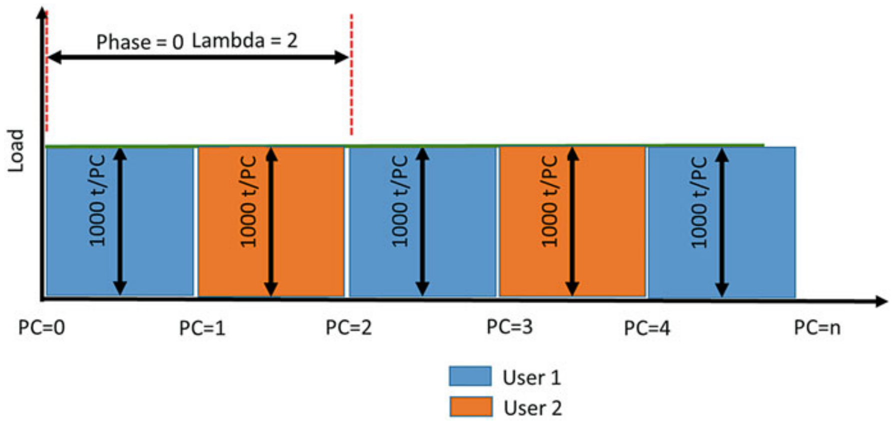


**Fig. 8.1** Depiction of simulation case 1

## 8.2 Simulation Case 2–2 Users

In this two-user case, one user submits a sine load, and the other user submits a cosine load. The sine-load[1] user submits tasks in a start–stop fashion. The cosine-load user does so as well, but with a phase shift that complements the sine-load. In the first case, $c_1$ submits a workload that is complementary but has a higher rate than that of $c_2$ by 100%. The configuration for the simulator in this case is as follows:

```
(1) simulator.setNumberOfSubmitters(2);
(2) simulation.setWavelength("2:2");
(3) simulation.setPhase("0:1");
(4) simulation.setAmplitude("1000:500");
(5) simulation.setTaskCount("5000:5000");
```

We create a setup with two submitters (1), each of which has a wavelength of 2 (2). The sine-load user submits a load at PC zero and then at even PCs. The cosine-load user submits a load at PC 1 and then at odd PCs (3). The actual loads are 1000 tasks/PC for $c_1$ and 500 tasks/PC for $c_2$ (4). Both users send the same number of tasks (5000) over the duration of the simulation (5) (Fig. 8.2).

[1]An un-shifted sine wave can go negative, which does not make sense in this context. For this reason, we only consider the positive when we talk about sine and cosine: $\max\left\{L_1(x, PC)\sin\left(\pi^* PC + \frac{3\pi}{2}\right), 0\right\}$

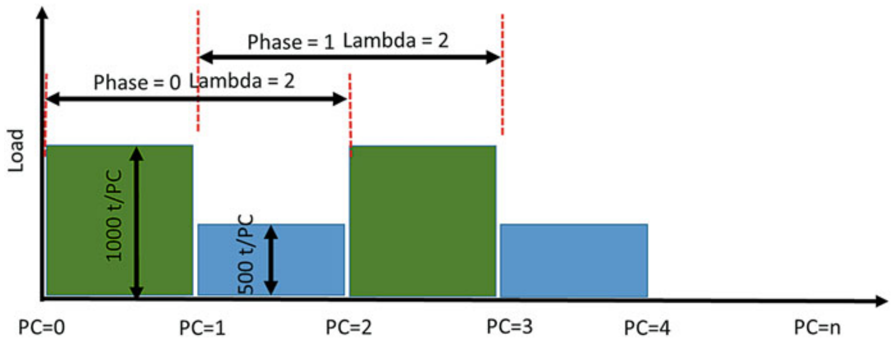


**Fig. 8.2** Depiction of simulation case 2

## 8.3 Simulation Case 3–2 Users

In this two-user case, one user submits a sine load and the other user submits a constant load. For a given lambda, the total number of tasks submitted by each user is the same. Furthermore, the total number of tasks submitted by each user during the simulation is the same: 5000 each.

The API level configuration is as follows:

```
    :
(1) simulator.setNumberOfSubmitters(2);
(2) simulation.setWavelength("0:2");
(3) simulation.setPhase("0:0");
(4) simulation.setAmplitude("100:200");
(5) simulation.setTaskCount("5000:5000");
    :
```

## 8.4 Simulation Case 4–2 Users

In this two-user case, one user first submits a large, one-time workload of 5000 tasks. Then, a second user submits a one-time workload of 50 tasks, a mere fraction of the total number of tasks submitted by the first user. Interestingly, a small workload can be overwhelmed in this type of case. Configuring the simulator for this simulation is accomplished by using the API as follows:

```
    :
(1) simulator.setNumberOfSubmitters(2);
(2) simulation.setWavelength("2:2");
(3) simulation.setPhase("0:1");
(4) simulation.setAmplitude("5000:50");
(5) simulation.setTaskCount("5000:50");
    :
```

## 8.5 Simulation Case 5–2 Users

This case is similar to case 4, but the second user submits only 1 task/PC until each of their 10 tasks have been submitted. The API level configuration is as follows:

```
    :
(1) simulator.setNumberOfSubmitters(2);
(2) simulation.setWavelength("2:2");
(3) simulation.setPhase("0:1");
(4) simulation.setAmplitude("5000:1");
(5) simulation.setTaskCount("5000:10");
    :
```

## 8.6 Simulation Case 6–6 Users

This six-user case resembles case 2, but 5 users each submit a sine load in equal parts and one user submits a cosine load. All of the users submit their tasks in a start–stop fashion. $c_1 = c_2 = c_3 = c_4 = c_5$, but $c_6$ submits a complimentary workload at a higher rate. The configuration is as follows:

```
(1) simulator.setNumberOfSubmitters(6);
(2) simulation.setWavelength("2:2:2:2:2:2");
(3) simulation.setPhase("0:0:0:0:0:1");
(4) simulation.setAmplitude("200:200:200:200:200:500");
(5) simulation.setTaskCount("1000:1000:1000:1000:1000:5000");
```

The setup includes six submitters (1), each with a wavelength of 2 (2). The sine-load user submits a load at PC zero and then at even PCs. The cosine-load user submits a load at PC 1 and then at odd PCs (3). For $c_1$. . $c_5$, the actual load is 200 tasks/PC up to 1000 tasks, and for $c_6$, the actual load is 500 tasks/PC up to 5000 tasks (4–5).

## 8.7 Simulation Case 7–6 Users

This six-user case resembles case 3, but 5 users each submit a constant load in equal parts and one user submits a sine load. For a given lambda, the total number of tasks submitted by the 5 constant-load users is the same as the total number of tasks submitted by the sine-load user.

The API level configuration is as follows:

```
     :
(1) simulator.setNumberOfSubmitters(6);
(2) simulation.setWavelength("0:0:0:0:0:2");
(3) simulation.setPhase("0:0:0:0:0:1");
(4) simulation.setAmplitude("20:20:20:20:20:200");
(5) simulation.setTaskCount("5000:5000:5000:5000:5000:5000");
     :
```

## 8.8 Simulation Case 8–6 Users

In this six-user case, 5 users submit large, one-time workloads of 1000 tasks. Then, one user submits a one-time workload of 50 tasks, a fraction of the total number of tasks submitted by the other users. This case resembles case 4, but with 6 users. The goal in this case is to analyze the multi-user case scenario of our workload. This was accomplished by using the API to configure the simulator as follows:

```
     :
(1) simulator.setNumberOfSubmitters(6);
(2) simulation.setWavelength("2:2:2:2:2:2:2");
(3) simulation.setPhase("0:0:0:0:0:0:1");
(4) simulation.setAmplitude("1000:1000:1000:1000:1000:50");
(5) simulation.setTaskCount("1000:1000:1000:1000:1000:50");
     :
```

## 8.9 Simulation Case 9–6 Users

This is similar to case 5, but with 6 users. 5 users each send 1000 tasks, while one user sends 1 task/PC. The API level configuration is as follows:

```
    :
(1) simulator.setNumberOfSubmitters(6);
(2) simulation.setWavelength("2:2:2:2:2:2");
(3) simulation.setPhase("0:0:0:0:0:1");
(4) simulation.setAmplitude("1000:1000:1000:1000:1000:1");
(5) simulation.setTaskCount("1000:1000:1000:1000:1000:10");
    :
```

## 8.10 Simulation Cases 10 and 14

Cases 10 and 14 are similar to case 6, but they feature 11 and 21 users, respectively.

## 8.11 Simulation Cases 11 and 15

Cases 11 and 15 are similar to case 7, but they feature 11 and 21 users, respectively.

## 8.12 Simulation Cases 12 and 16

Cases 12 and 16 are similar to case 8, but they feature 11 and 21 users, respectively.

## 8.13 Simulation Cases 13 and 17

Cases 13 and 17 are similar to case 9, but they feature 11 and 21 users, respectively.

## 8.14 Assumptions

While various assumptions are made in the aforementioned cases, each supports the overall goal of assessing the feasibility of a new scheduling model that uses three system parameters to reduce time-in-system. Due to the complexity of the task at hand, a number of assumptions are made. These assumptions are enumerated in this section.

### *8.14.1 No-Randomness Assumption*

The simulation runs feature no randomness: a given input *always* led to the same output, and no random parameters or variables appeared anywhere in the simulation. This is required as the interest is in achieving predictable improvements in performance and not statistical significance across the various results. The simulation and results were predictable and repeatable in every case. The applicability of the Rawlsian Fair scheduler hypothesis is tested in each simulation. For each simulation, (a) the submission profiles are varied, (b) the number of users are varied, and (c) the method used to calculate seniority are varied. As the interest is to test the hypothesis under specific conditions, the specificity of the scenario will prove to be more applicable in a controlled environment and without any randomness as to the submission profile or number of users.

### *8.14.2 Task-Based-Workload Assumption*

In the cases considered, a given job is composed of many requests or tasks, each of which is self-contained and can be executed independently. In addition, resources are allocated dynamically. Due to the nature of the interactive workloads that run in HPC environments, the resources allocated to a given user can change with every scheduling cycle. We also assume that the tasks are malleable (Feitelson et al. 1997): they are able to take advantage of new resources allocated during the scheduling cycle.

### *8.14.3 Resource-Constraint Assumption*

The number of resources is held constant at 100 CPUs in all of the simulations. Although the number of resources is largely irrelevant, it had to be smaller than the workload and to remain constant. The number of resources had to be smaller than the workload because if it was not, users would not have been forced to compete for

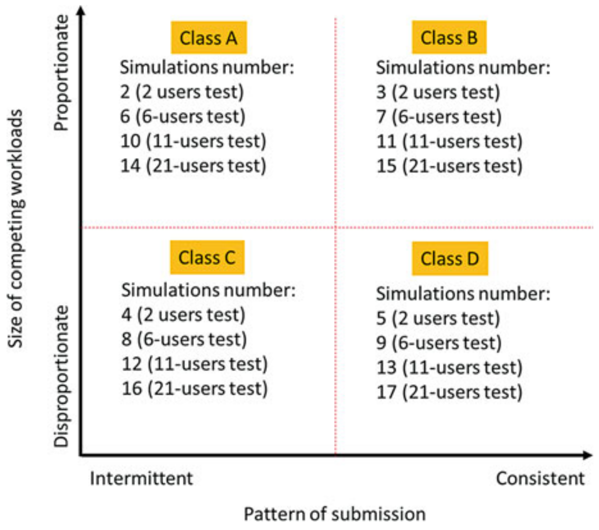


**Fig. 8.3** Classification of the 16 simulation scenarios

resources and the simulation would not have been representative of real-world cases. The number of resources was held constant to reduce the number of variables.

### *8.14.4 Simulation Parameters and Configuration*

17 simulation cases were run, each with seven scheduling algorithms: traditional fair-share and Rawlsian Fair with 6 different BH values. Simulation 1 was a "smoke-test" to assess the basic functionality of the simulator, but the other 16 tests were broken into four different classes of simulations, as is described in Sect. 1.2. Figure 8.3 maps the simulation runs to the four classes of job profiles. Below, an overview is provided of all of the simulations and the expectations for each class of simulation.

#### 8.14.4.1 Simulations for Class-A: Simulations 2, 6, 10 and 14

Simulations 2, 6, 10, and 14 were conducted to determine the applicability of the Rawlsian Fair algorithm vis-à-vis Class A workloads. In these simulations, the overall load was held constant at 10,000. The parameters of the simulation keeps one user's submission load profile constant while distributing other loads among the rest of the users. The goal of these tests was to determine whether Rawlsian Fair is *as good as* fair-share scheduling. Rawlsian Fair was expected to introduce no addition delays, and the simulations validated this assumption. Since fair-share scheduling seeks long-term fairness (Kleban and Clearwater 2003), the primary objective of these tests was to make sure that Rawlsian Fair does not generate more delays than fair-share.

#### 8.14.4.2 Simulation Class-B: Simulations 3, 7, 11, and 15

Simulations 3, 7, 11, and 15 were conducted to determine the applicability of the Rawlsian Fair algorithm vis-à-vis Class B workloads. In each of these simulations, one or more of the workloads submitted is a constant workload. In each case, the constant workload(s) competes with intermittent workloads. Since a constant workload submits a constant number of tasks, in 2-user simulations, the results achieved by the Rawlsian Fair algorithm were expected to be *as good as* those achieved by fair-share algorithm. As the number of users increased, the Rawlsian Fair algorithm was expected to yield results that were increasingly better than those of fair-share scheduling.

#### 8.14.4.3 Simulation Class-C: Simulations 4, 8, 12, and 16

Simulations 4, 8, 12, and 16 were conducted to determine the applicability of the Rawlsian Fair algorithm vis-à-vis Class C workloads. In each simulation, Rawlsian Fair scheduling was expected to reduce the delay experienced by the smallest user(s).

#### 8.14.4.4 Simulation Class-D: Simulations 5, 9, 13, and 17

Simulations 5, 9, 13, and 17 were conducted to determine the applicability of the Rawlsian Fair algorithm vis-à-vis Class D workloads. In each simulation, Rawlsian Fair scheduling was expected to reduce the delay experienced by the smallest user(s). As the number of users increased from 2 to 21, the delay experienced by the smallest user(s) was expected to remain constant.

# Chapter 9
# Overview of Results

Chapter 8 outlined 17 cases simulated using dSim. Both the traditional, fair-share scheduling algorithm and the Rawlsian Fair scheduling algorithm developed for our work were used in these simulations. Table 9.1 outlines the different classes of simulations and their purposes.

## 9.1 Results Recap

The results reveal that the Rawlsian Fair scheduling algorithm both significantly decreased the times-in-system of the users with the lowest relative numbers of tasks and decreased the time-in-system variance for every user. This section offers an overview of the delays incurred by the users in the simulations. Data analysis is conducted per class of requests, as is outlined in Sect. 1.2. Data analysis for the Least Benefited User (LBU) will be considered as defined in Rawls (2001).

The Rawlsian Fair scheduling algorithm uses seniority as well as the traditional parameters of load and priority. Seniority is updated by the scheduler according to the changes that occur as tasks enter and leave the system. Seniority is included to decrease the time-in-system experienced by users with low numbers of tasks. In fair-share scheduling, the delay is normalized across the users. This effect is revealed in our results. In Rawlsian Fair scheduling, however, seniority moves tasks to the front of the queue: the tasks in the queue are ordered first by seniority and then by arrival time, so that of the tasks that share the highest seniority value, the first one in the queue is executed first. As mentioned previously, the Rawlsian Fair scheduler uses buckets to calculate seniority. Seniority is calculated based on the current state of the bucket (i.e. the number of tasks in the bucket), and all of the tasks in a given bucket are finished before the scheduler moves on to the next bucket (Fig. 9.1).

The Placement Counter (PC) dictates the target bucket location: PC = 1 indicates that the tasks are destined for Bucket 1, PC = 2 indicates that the tasks are destined for Bucket 2, and so on. Bucket History (BH) size is another variable. Seniority is

A. Sedighi, M. Smith, *Fair Scheduling in High Performance Computing Environments*, https://doi.org/10.1007/978-3-030-14568-2_9

**Table 9.1** Summary of simulation types and expected outcome

| Workload classification | Relevance |
|---|---|
| Class A | Intermittent workloads occur when users submit jobs throughout the day. These simulations describe situations in which complementary users of different load compete for resources. It considers how performance is affected as the number of users increase. |
| Class B | These simulations describe situations in which steady workloads compete for resources with intermittent workloads. |
| Class C | These simulations describe situations in which one user is competing for resources for a one-time burst submission against larger workloads. |
| Class D | These simulations describe situations in which small but steady workloads compete with larger workloads. |

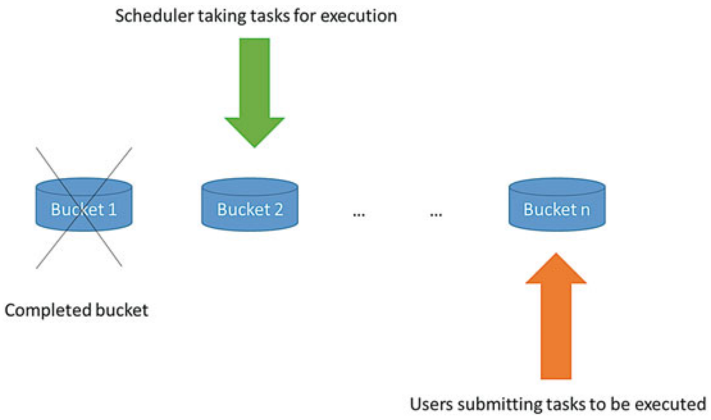

Scheduler taking tasks for execution
Bucket 1
Bucket 2
...
...
Bucket n
Completed bucket
Users submitting tasks to be executed

**Fig. 9.1** Illustration of the role of buckets in placing (via the Placement Counter) and executing tasks

calculated across a number of buckets denoted by the BH value. BH = 1 (Fig. 9.1) means that seniority is calculated based on the parameters of the current bucket, i.e. the oldest bucket waiting to be drained.

Once all of the tasks in a given bucket have been scheduled for execution, the bucket is deleted and the scheduler moves on to the next bucket. In each of the simulations, the number of processing nodes (or CPUs) was held constant at 100, meaning that 100 tasks could be processed simultaneously. Each task takes 1000 ms to complete, so if 1000 tasks were submitted and the system was able to process 100 at a time, it would take 10 s to complete all of the tasks in a real-world scenario.

In BH = 2 scenarios (Fig. 9.2), the tasks are placed in their respective buckets, but seniority is calculated based on the parameters of two buckets. In other words, the total number of pending tasks and the total number of submitted tasks are based on the values spanning two buckets.

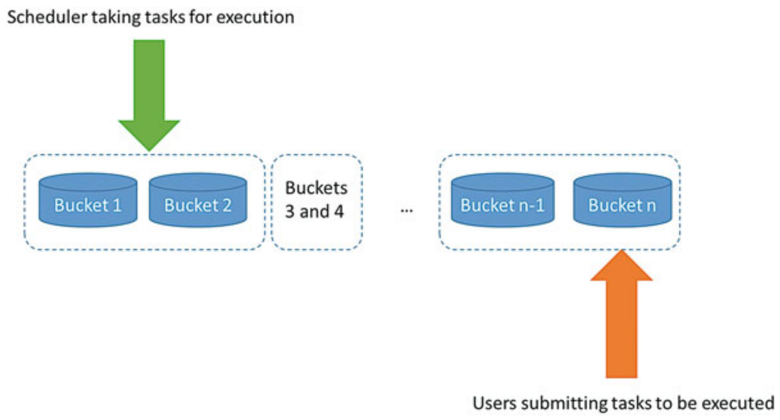

Scheduler taking tasks for execution
Bucket 1
Bucket 2
Buckets 3 and 4
...
Bucket n-1
Bucket n
Users submitting tasks to be executed

**Fig. 9.2** Scheduling scenario with BH = 2

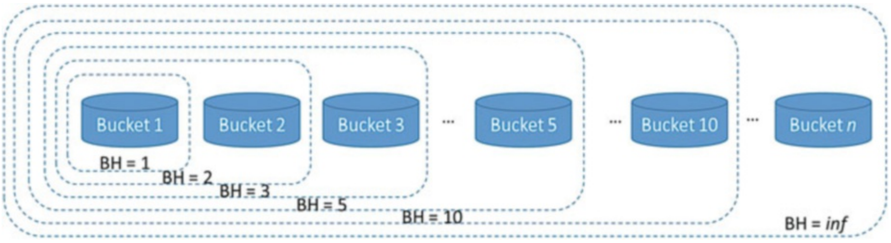


**Fig. 9.3** Calculating seniority based on BH value

Simulations were run for different workloads and six different BH sizes:

$$BH = \{1, 2, 3, 5, 10, inf\} \tag{9.1}$$

*Bh* = *inf* is a special case in which seniority is calculated by taking into consideration the entire submission history of a given user (Fig. 9.3). This scenario consider a system never forgets. While smaller BH values can suffer from local maxima problems (see the results of simulations 3 and 4), local maxima problems are resolved when *Bh* = *inf*. A side effect, however, is that disproportionately large jobs tend to suffer.

Although the aim of the Rawlsian Fair scheduler is to improve the fairness of the system, as the FUD conjecture suggests, one of the other parameters will suffer as a result of this change.

Our results revealed that although the Rawlsian Fair scheduler improved fairness of the system, i.e. smaller users experienced smaller times-in-system—the overall dynamicity of the system decreased. To calculate seniority, the incoming tasks were placed in buckets (or queues) (see Sect. 6.2) in the order in which they arrived. In cases in which multiple users had the same seniority level (e.g. in simulation 6), the

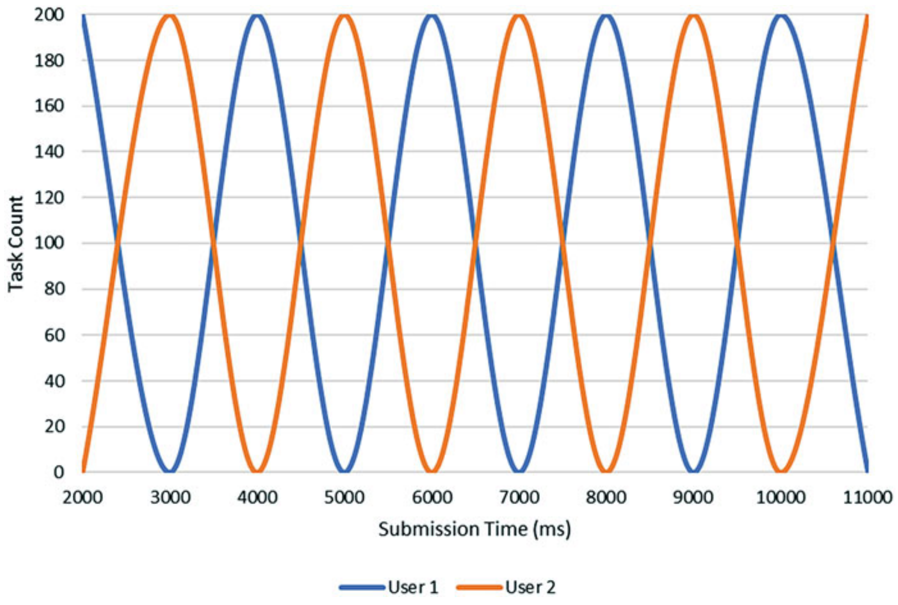


**Fig. 9.4** Task submission pattern for simulation 1

tasks were executed in a first-come-first-served fashion. The same applies for users with the same number of overall tasks, as one would expect. In cases in which two or more users submitted the same number of tasks at the same time, the tasks were executed in the order in which they arrived in the queue.

### *9.1.1 Simulation 1 Analysis and Results*

Simulation 1 was a "smoke simulation" to make sure that our system worked as expected end-to-end. Two users submitted 1000 tasks each at 200 tasks/PC (Fig. 9.4).

Each user submitted their next task after the other user had submitted theirs, creating a sine–cosine submission pattern. We expected that the delays incurred for each task would be similar for Rawlsian Fair with BH = *inf* and fair-share (Figs. 9.5, 9.6).

First, we compare the simulation results for fair-share and Rawlsian Fair with a bucket history of infinity (*inf*). Rawlsian Fair's stepwise task-delay line is the result of the bucketing feature for task submission. Tasks are sorted into buckets and executed in the order in which they arrive. There were 100 worker nodes, so user 1's first 100 tasks were sorted into bucket 1, and so on.

The task delays obtained with each scheduling method were complimentary because there were only two users: an increase in the delay experienced by one user meant a decrease in the delay experienced by the other (Fig. 9.7).

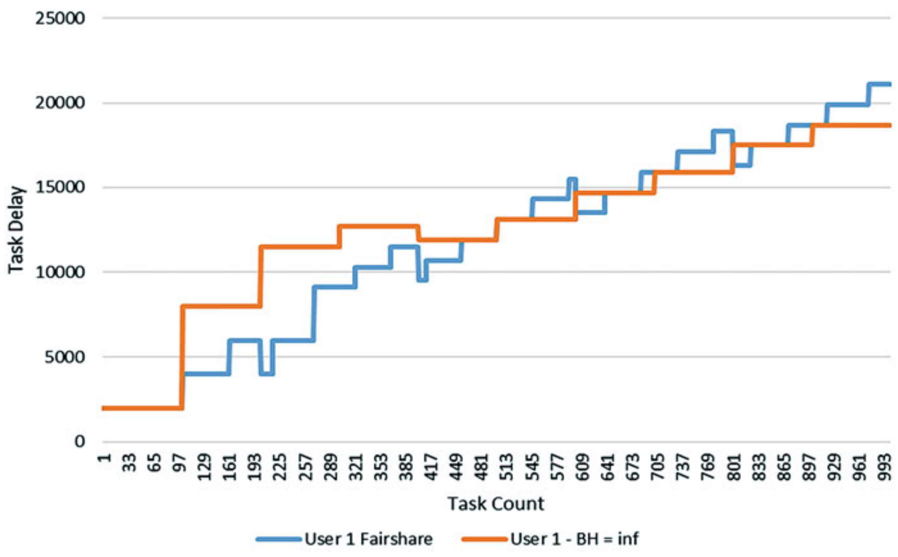


**Fig. 9.5** User 1's task delay in simulation 1 with BH = *inf*

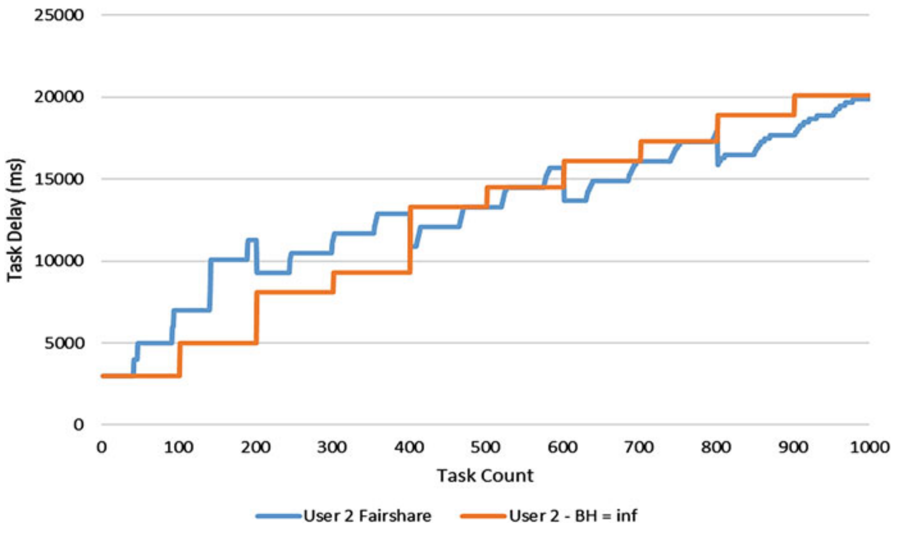


**Fig. 9.6** User 2's task delay in simulation 1 with BH = *inf*

The goal in this simulation was to evaluate how close Rawslian Fair scheduling gets to traditional fair-share when two users share the same job-submission profile. The results for all of the BH sizes in simulation 1 are also presented (Figs. 9.8, 9.9), and they show a similar delay pattern.

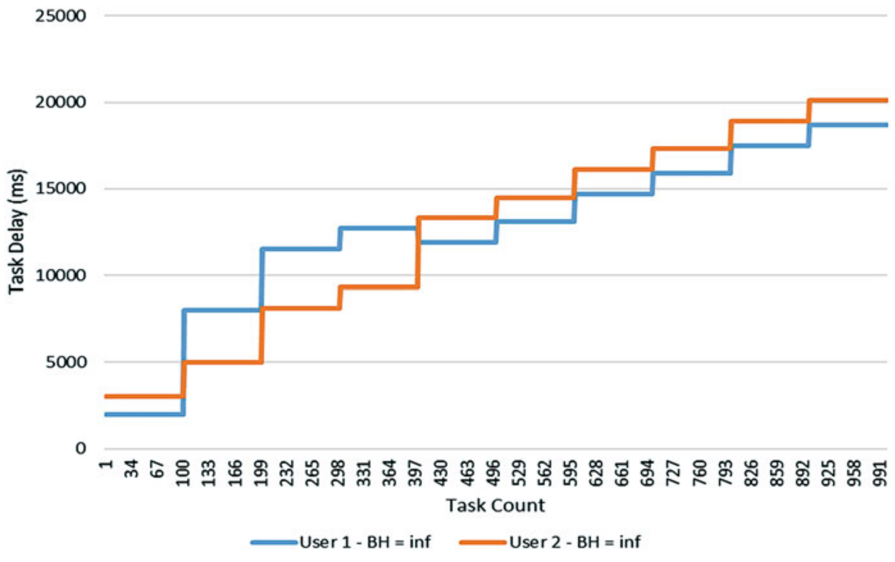


**Fig. 9.7** Task delay comparison of both users for simulation 1

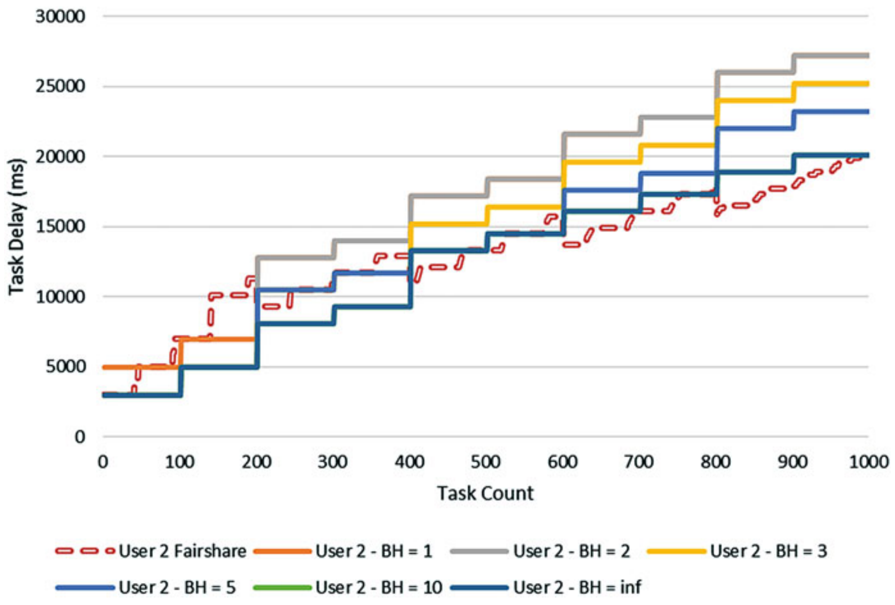


**Fig. 9.8** User 2's task delay for all BH sizes in simulation 1

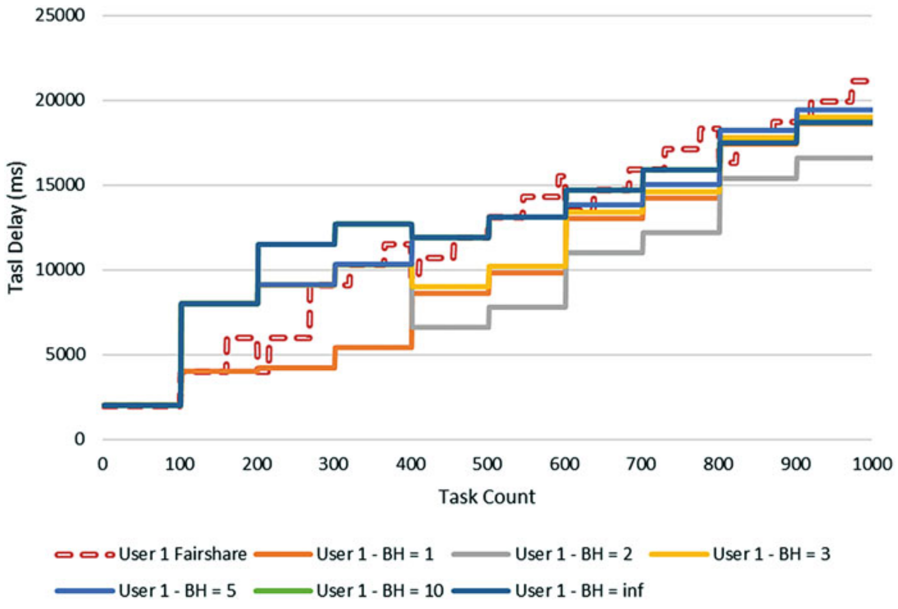


**Fig. 9.9** User 1's task delay for all BH sizes in simulation 1

## 9.2 Lessons Learned

In the calculation of seniority, incoming tasks were timeboxed into "buckets" (see Sect. 6.2) denoted by the placement counter (PC) variable. The granularity of the placement counter had a profound effect on the performance levels of the less senior tasks. While the PC granularity had to be set at 1000 ms (1 s) (equal to the task-execution size) for seniority to be calculated, this had the side effect of delaying less senior tasks by their position within each bucket. This delay resulted from the action of the First-Come-First-Serve (FCFS) algorithm, which was used to take equal-seniority tasks off the queue. Even though the tasks were submitted at the "same time," even a fraction of a second can affect the arrival time. dSim is aware of communication delays and uses them in setting the arrival time of each task.

Regardless of the PC value, however, Rawlsian Fair scheduler performed equal to or better than fair-share overall for Class A and Class B workloads. A comparison of the results of simulations 5 and 9 further demonstrates this point. While the number of tasks was the same in both simulations, simulation 9 had more users. As the number of users increased, so did the effectiveness of Rawlsian Fair.

Rawlsian Fair performed universally better with Class C and Class D workloads. This is apparent when simulation 5 is compared to simulations 13 and 17. The tasks submitted by a single, the large user for simulation 5, are distributed among 5 users in simulation 9, 11 users in simulation 13, and 20 users in simulation 17. The outcome of each simulation is the same for the smallest user, however, demonstrating the effect of Rawlsian Fair's use of seniority.

The number of tasks distributed among the users is held constant across each of the simulations, including simulations 10 and above, and the effect of Rawlsian Fair becomes increasingly apparent as the number of users increases. Seniority consistently benefits disproportionately small users, but because Rawlsian Fair picks smaller users from sets of larger users, this effect becomes more pronounced as the number of users increases.

## 9.3 Simulation Results and Analysis

The following chapters review the results of the various simulation runs. We generated over 600,000 data points,[1] and we illustrate our key findings. Given the sheer number of data points, every data point cannot be independently considered.

Some of the simulation results are grouped into pairs because while the number of tasks is the same for the members of each pair, one member of each pair demonstrates the effect of Rawlsian Fair on the smallest user, and the other member demonstrates the effect of seniority as the number of users increases.

[1] 17 tests, 7 different scheduling algorithms, and 5000+ tasks per run

# Chapter 10
# Class A Results and Analysis

## 10.1 Class A Simulations

Class A simulations describe situations featuring intermittent workloads. In Simulation2, two users submit equal but complementary workloads. As the number of users increases—to 6 in Simulation6, 11 in Simulation10, and 21 in Simulation14—one user's submission rate stays constant while the workload from the second user (5000 tasks) is distributed among the rest of the users in the system. In Simulation6, users 1–5 submit 1000 tasks each and 5000 total, while user 6 submits 5000 tasks on their own. In Simulation10, users 1–10 submit 500 tasks each, while user 11 submits 5000 tasks on their own. In Simulation14, users 1–20 submit 250 tasks each, while user 21 submits 5000 tasks on their own.

## 10.2 Class A Results

When the workloads are equal but complementary, as in Simulation2, a difference in performance of less than 2% is observed between fair-share and Rawlsian Fair scheduling methods. As the gap between the largest user and the smallest user increases, however, Rawlsian Fair produces an average performance that is approximately 80% better than that of fair-share (Table 10.1). The average total delay remains constant, confirming the fairness of Rawlsian Fair.

### *10.2.1 Simulation 2 Analysis and Results*

In simulation 2, two users submitted different loads. User 1 submitted 1000 tasks/PC up to a total of 5000 tasks, and user 2 submitted 500 tasks/PC up to a total of 5000

**Table 10.1** Class A Submission type average delay

| Class A average delay | | | | |
|---|---|---|---|---|
| | Simulation 2 | Simulation 6 | Simulation 10 | Simulation 14 |
| User under test | User 1 | Users 1–5 | Users 1–10 | Users 1–20 |
| Fairshare average delay (ms) | 62,891 | 61,824 | 66,873 | 73,617 |
| Average Rawlsian Fair delay (ms) | 61,455 | 34,783 | 34,783 | 34,783 |
| Rawlsian Fair delay per user (ms) | 61,455 | User 1: 29,321 | User 1: 28,641 | User 1:28,641 |
| BH = *inf* | | User 2: 32,041 | User 2: 30,001 | User 2:28,641 |
| | | User 3: 34,761 | User 3: 38,161 | User 3:31,361 |
| | | User 4: 37,481 | User 4: 39,521 | User 4:32,721 |
| | | User 5: 40,311 | User 5: 41,101 | User 5:34,081 |
| | | | User 6: 34,081 | User 6:35,441 |
| | | | User 7: 35,441 | User 7:30,001 |
| | | | User 8: 36,801 | User 8:30,001 |
| | | | User 9: 32,721 | User 9:31,361 |
| | | | User 10:31,361 | User 10:41,101 |
| | | | | User 11:39,521 |
| | | | | User 12:39,521 |
| | | | | User 13:38,161 |
| | | | | User 14:38,161 |
| | | | | User 15:36,801 |
| | | | | User 16:36,801 |
| | | | | User 17:35,441 |
| | | | | User 18:34,081 |
| | | | | User 19:32,721 |
| | | | | User 20:41,101 |

(continued)

**Table 10.1** (continued)

| Class A average delay | | | | |
|---|---|---|---|---|
| | Simulation 2 | Simulation 6 | Simulation 10 | Simulation 14 |
| Average improvement | <2% | 86% | 95% | 114% |
| Improvement per user | <2% | User 1: 118% | User 1: 133% | User 1: 157% |
| | | User 2: 99% | User 2: 123% | User 2: 157% |
| | | User 3: 84% | User 3: 75% | User 3: 135% |
| | | User 4: 70% | User 4: 69% | User 4: 125% |
| | | User 5: 58% | User 5: 63% | User 5: 116% |
| | | | User 6: 96% | User 6: 108% |
| | | | User 7: 89% | User 7: 145% |
| | | | User 8: 82% | User 8: 145% |
| | | | User 9: 104% | User 9: 135% |
| | | | User 10: 113% | User 10: 79% |
| | | | | User 11: 86% |
| | | | | User 12: 86% |
| | | | | User 13: 93% |
| | | | | User 14: 93% |
| | | | | User 15: 100% |
| | | | | User 16: 100% |
| | | | | User 17: 108% |
| | | | | User 18: 116% |
| | | | | User 19: 125% |
| | | | | User 20: 79% |

tasks. The number of available CPU's was set at 100 for the duration of the simulation (Fig. 10.1).

Simulation 2 featured 2 users and was run for all BH options and fair-share (Figs. 10.2 and 10.3).

The red line traversing both figures represents the task delay incurred using the fair-share scheduler. As expected, fair-share normalized the delay for both users, irrespective of their different submission profiles. Very little difference was observed at the micro-level between fair-share and Rawlsian Fair, but the results are interesting nonetheless. As Fig. 10.1 shows, the submission is interweaving amongst the two users. As such, each bucket contains tasks from only one user. User 2's tasks came in at a lower rate, and with varying of the BH sizes, user 2's tasks were lowered task delay to start (red circle in Fig. 10.3). This was particularly true for BH = inf, at which user 2, who had a lower task-submission rate, started off with a lower task delay. This, in turn, caused user 1's tasks to fall behind, at which point they were

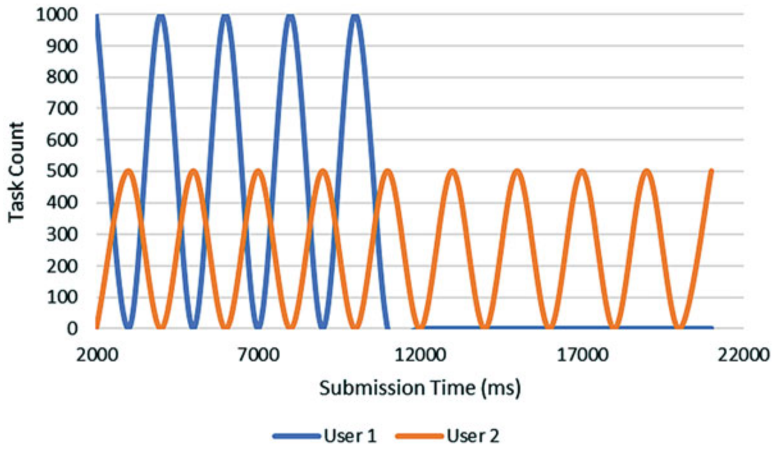


**Fig. 10.1** Task submission for simulation 2

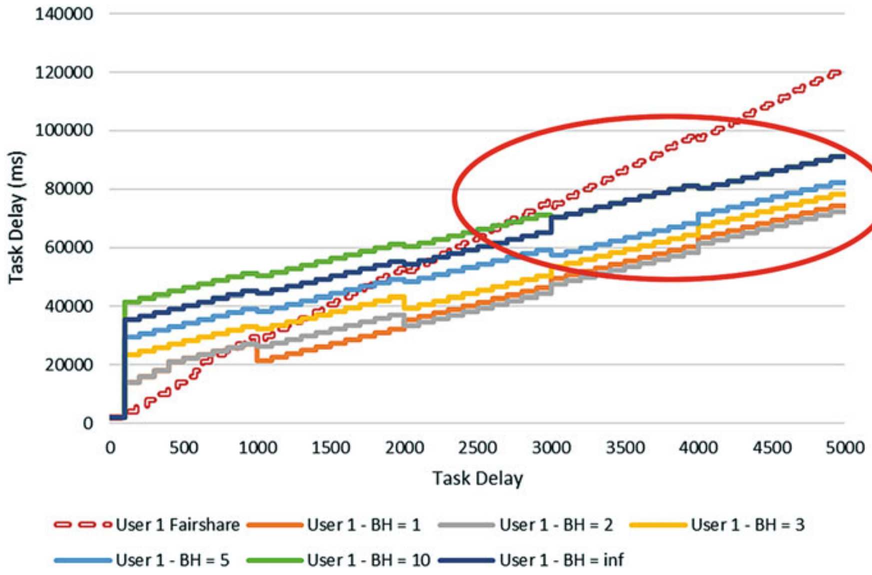


**Fig. 10.2** User 1’s task delay for all BH sizes in simulation 2

assigned greater seniority and were executed ahead of user 2’s tasks (as is indicated by the red circle in Fig. 10.3).

High BH values, such as 10 or *inf*, were the best-case scenarios. In other cases, user 2’s tasks lagged in execution behind user 1’s because of how they were sorted into buckets. Once each user had submitted all of their 5000 tasks, the FCFS aspect of the scheduler took over. Table 10.2 compares the results obtained with Rawlsian Fair and fair-share in simulation 2.

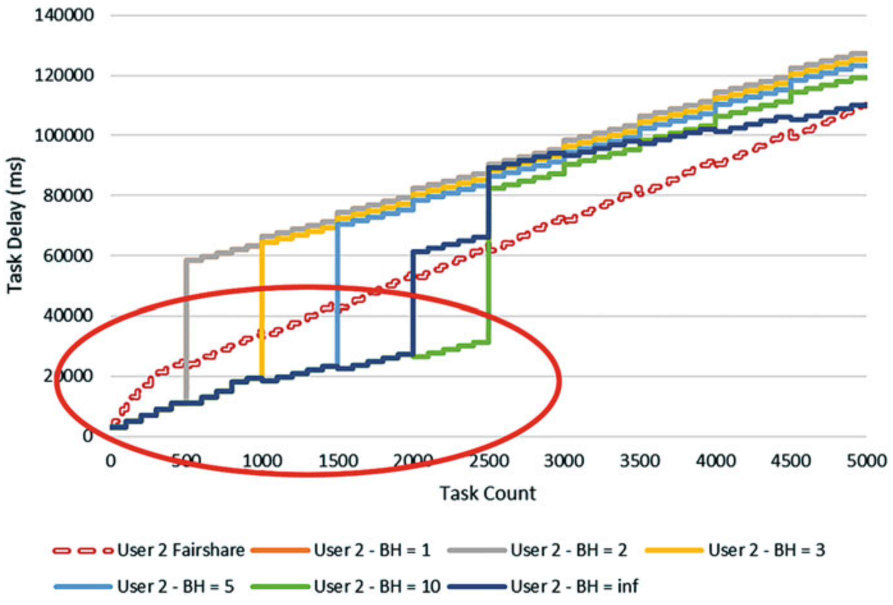


**Fig. 10.3** User 2's task delay for all BH sizes in simulation 2

**Table 10.2** Comparison of the delays generated by Fair-Share (FS) and Rawlsian Fair (RF) in simulation 2

| | FS delay (ms) | RF delay (ms) BH = 1 | RF delay (ms) BH = 2 | RF delay (ms) BH = 3 | RF delay (ms) BH = 5 | RF delay (ms) BH = 10 | RF delay (ms) BH = inf |
|---|---|---|---|---|---|---|---|
| User 1 | 62,891 | 42,133 | 41,913 | 47,955 | 53,615 | 64,935 | 61,455 |
| vs FS | N/A | 49% | 50% | 31% | 17% | −3% | 2% |
| User 2 | 61,824 | 84,221 | 84,221 | 78,069 | 72,079 | 60,099 | 63,029 |
| vs FS | N/A | −27% | −27% | −21% | −14% | 3% | −2% |

### *10.2.2 Simulation 6 Analysis and Results*

In simulation 6, one user submitted 5000 tasks and five other users submitted 1000 tasks each (Fig. 10.4). The difference between simulation 6 and simulation 2 is that in simulation 6, users 1–5 submitted simultaneously at a rate of 200 tasks for every two PC ticks. In Simulation 6, the performance level that users 1–5 experienced under Rawlsian Fair with a minimum BH value of 1 (Table 10.3) exceeded the performance level they experienced under fair-share by as much as 170%. This was mainly because with BH = 1, only the size of the immediate bucket was taken into account. The remainder of this section compares the performance levels of Rawlsian

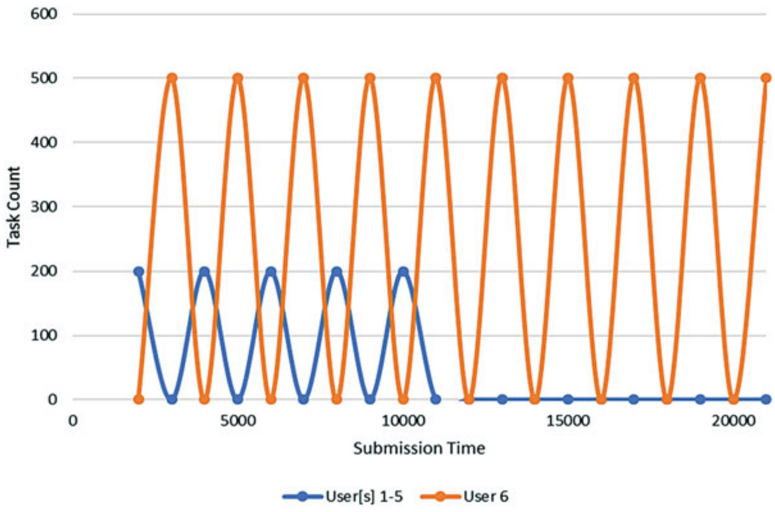


**Fig. 10.4** The task submissions of all 6 users in simulation 6

**Table 10.3** Comparison of the delays generated by Fair-Share (FS) and Rawlsian Fair (RF) in simulation 6

| | FS delay (ms) | RF delay (ms) BH = 1 | RF delay (ms) BH = 2 | RF delay (ms) BH = 3 | RF delay (ms) BH = 5 | RF delay (ms) BH = 10 | RF delay (ms) BH = inf |
|---|---|---|---|---|---|---|---|
| User 1 | 63,883 | 23,481 | 27,821 | 31,421 | 35,041 | 29,321 | 29,417 |
| vs Fairshare | N/A | 172% | 130% | 103% | 82% | 118% | 121% |
| User 6 | 62,635 | 85,891 | 87,091 | 83,381 | 83,271 | 90,251 | 89,701 |
| vs Fairshare | N/A | −27% | −28% | −25% | −25% | −31% | −30% |

Fair and fair-share for users 1–5, and again for user 6. The other comparison is amongst users 1–5 for a given simulation parameter.

Figure 10.6 shows the task delays that user 1 experienced with the different BH values. User 1 submitted their tasks first, followed by user 2, and so on. This is important because with time, the FCFS aspect of the Rawlsian Fair scheduler negatively affects tasks with the same seniority. Every user was still better off than they would have been under fair-share, but user 1's tasks were completed first (Fig. 10.5). Rawlsian Fair pushed out user 6 in favor of the smaller users (users 1–5). This feature delayed user 6, which is apparent in Fig. 10.7. In every case, the dotted red line represents the task delay generated by the Fair-Share scheduler.

Table 10.3 shows the different performance levels that user 1 experienced as the BH value varied. User 6, the largest user, experienced a level of performance that

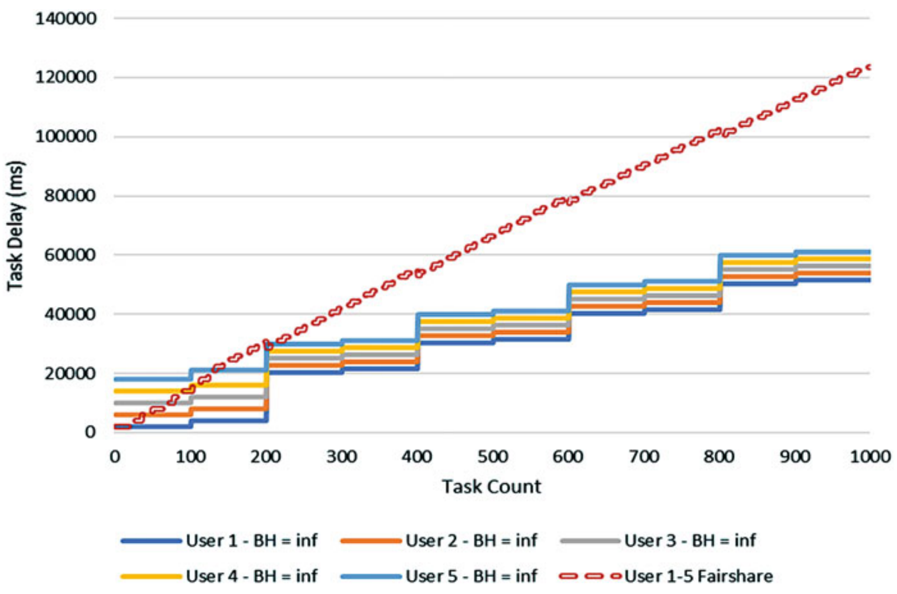


**Fig. 10.5** The task delays experienced by all users with BH = *inf* and Fair-Share in simulation 6

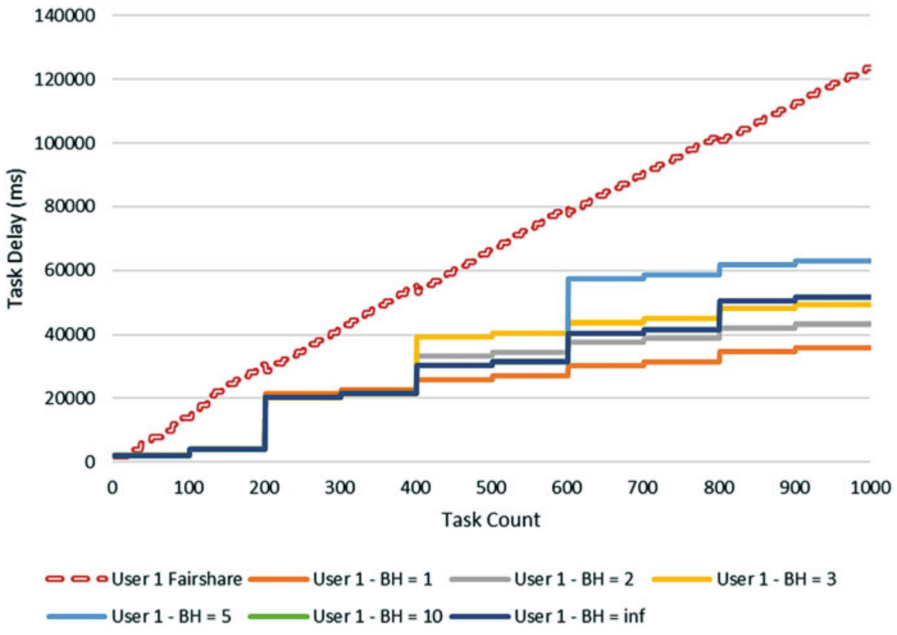


**Fig. 10.6** User 1's task delays with Fair-Share and all BH values in simulation 6

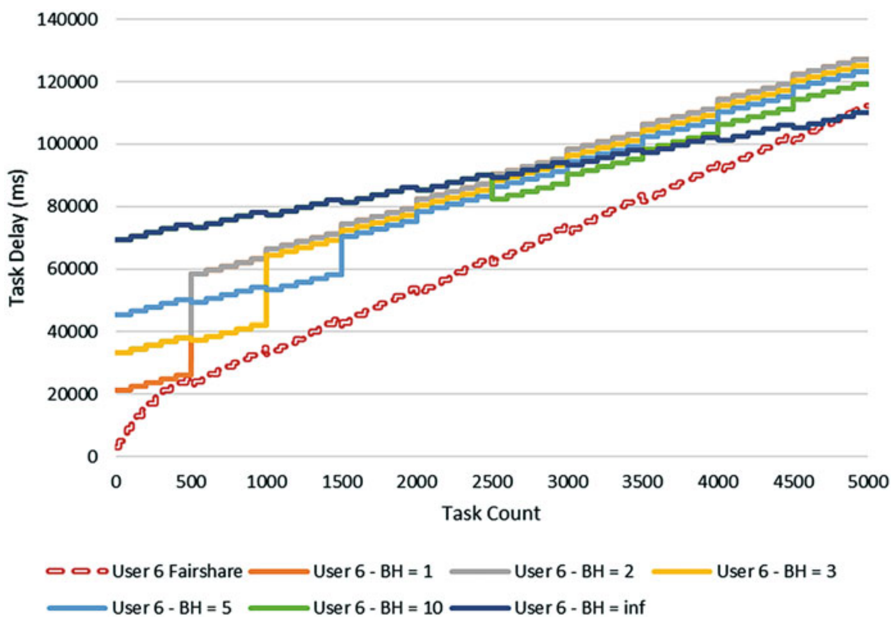


**Fig. 10.7** User 6's task delays with Fair-Share and all BH values in simulation 6

was expectedly worse than those of the other users because users 1–5 were given precedence by the Rawlsian Fair scheduler.

### *10.2.3 Simulations 10 and 14 Results and Analysis*

Simulation 10 featured 11 users, and simulation 14 featured 21 users. The total number of tasks submitted was the same in both simulations, but the number of tasks each user submitted decreased as the number of users increased. In simulation 10, users 1–10 submitted 100 tasks every two PCs up to a total of 5000 tasks, while user 11 submitted 500 tasks every two PCs up to a total of 5000 tasks (Fig. 10.8). In Simulation14, users 1–20 each submitted 50 tasks every two PCs, while user 21 submitted 500 tasks at a time up to a total of 5000 tasks (Fig. 10.10). In simulation 10, users 1–10 submitted their tasks to the same bucket simultaneously, starting with bucket 2. As a result, bucket 2 held 100 tasks from each of ten users, for a total of 1000 tasks. Users 1–20 in simulation 14 submitted their tasks in the same way: each submitted 50 tasks at a time, so bucket 2 held 1000 tasks. In both simulations, bucket 3 held 500 tasks submitted by the larger user.

In both simulations, the larger user—user 11 in simulation 10 and user 21 in simulations 14—experienced delays with Rawlsian Fair that exceeded the delays they experienced with traditional fair-share. The same pattern was observed in simulation 6. The delays the larger users incurred in both cases were almost identical

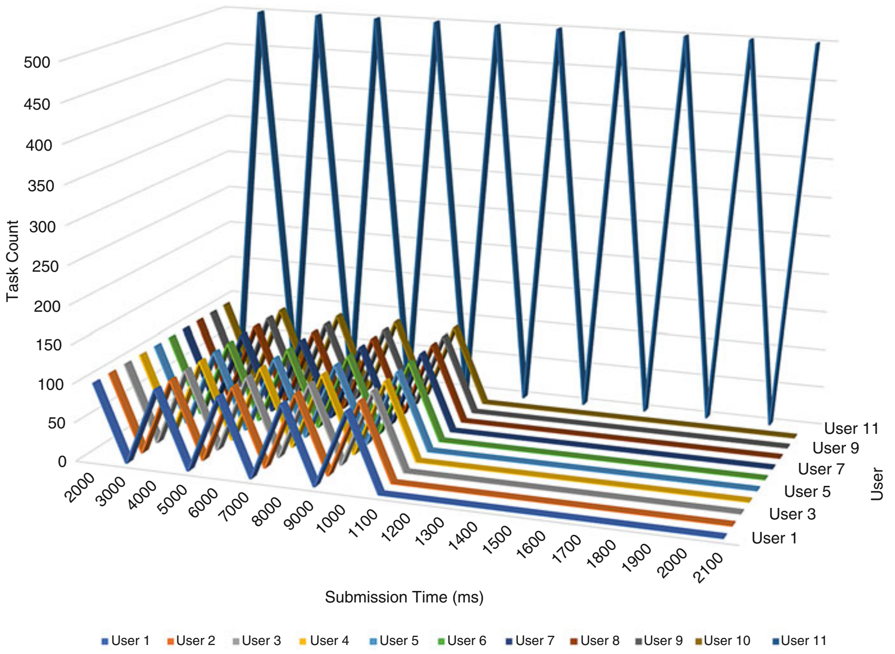


**Fig. 10.8** The task submissions of all 11 users in simulation 10

(Figs. 10.13 and 10.14) because in both cases, the scheduler did not send the larger user's task for execution until all of the smaller users' tasks had been executed. Figure 10.12 shows a close-up of the of one of the smaller users in simulation 10 (user 1), and Fig. 10.11 shows a close-up of the of one of the smaller users in simulation 14. Rawlsian Fair scheduling was able to pick this smaller user from the others in the system and execute their tasks earlier than Fair-Share would have. The different Bucket-History (BH) sizes affected the task delay only after each user's submission number had exceeded the minimum number of buckets required for BH to be valid—with BH = 3, there must be at least three buckets, and so on. The red circle in Fig. 10.12 shows this effect in simulation 10, and the red circle in Fig. 10.11 shows this effect in simulation 14. The delays incurred by the smaller users were due largely to the number of available resources (100 CPU's), which determined the number of tasks that could be executed simultaneously. This is apparent in Fig. 10.9 (for Simulation10) and Fig. 10.15 (for Simulation14), which show the task delays experienced by all of the users. The task delays experienced by the smaller users are similar because the scheduler picked the largest seniority in the order of arrival before moving on to the next user on a first-come-first-served basis.

Table 10.4 compares the performance characteristics of one of the smaller users (user 1) to those of the larger user (user 11). With every BH value in simulation

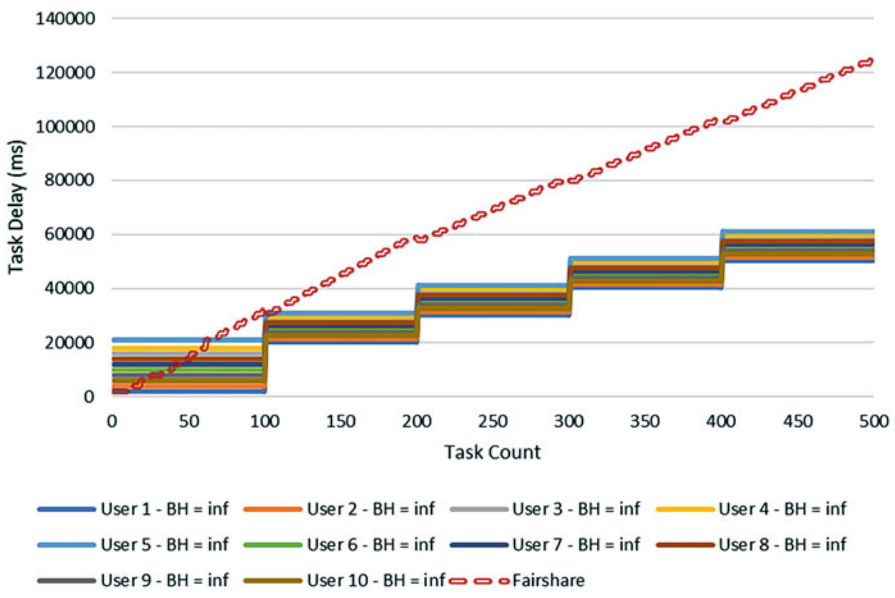


**Fig. 10.9** The task delays experienced by users 1–10 with BH = *inf* in simulation 10

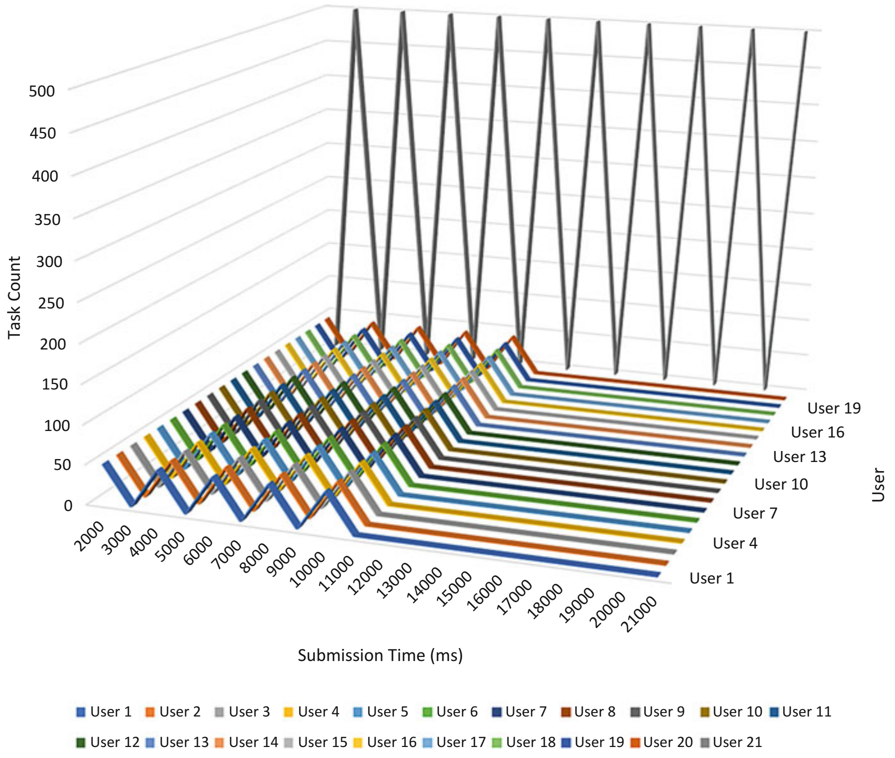


**Fig. 10.10** The task submissions of all 21 users in simulation 14

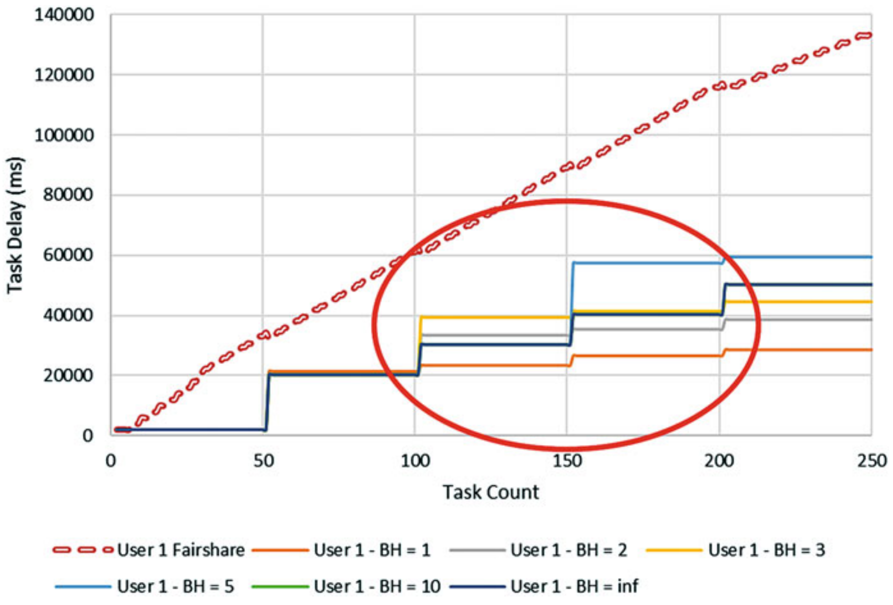


**Fig. 10.11** Task delay for user 1 for all BH values for simulation 14

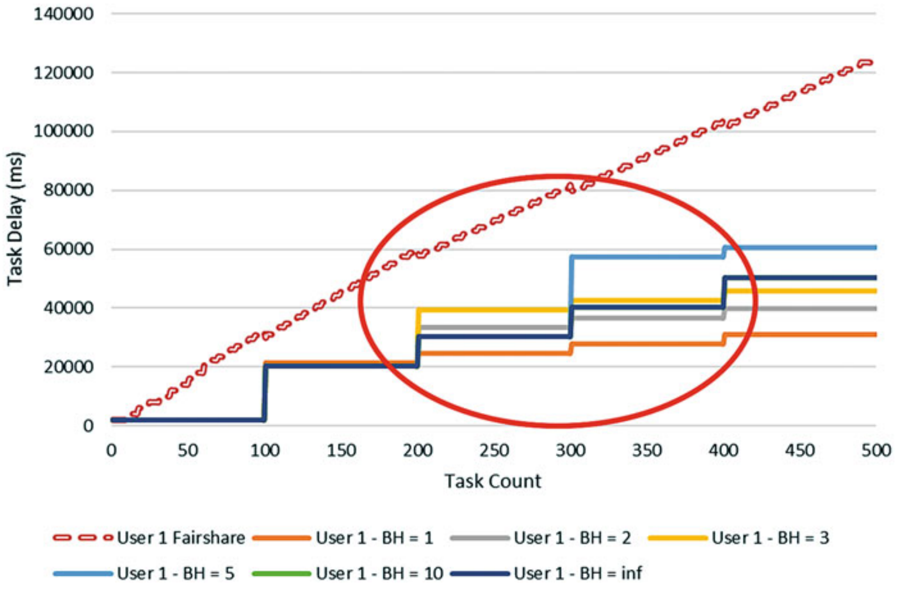


**Fig. 10.12** User 1’s task delays with all BH values in simulation 10

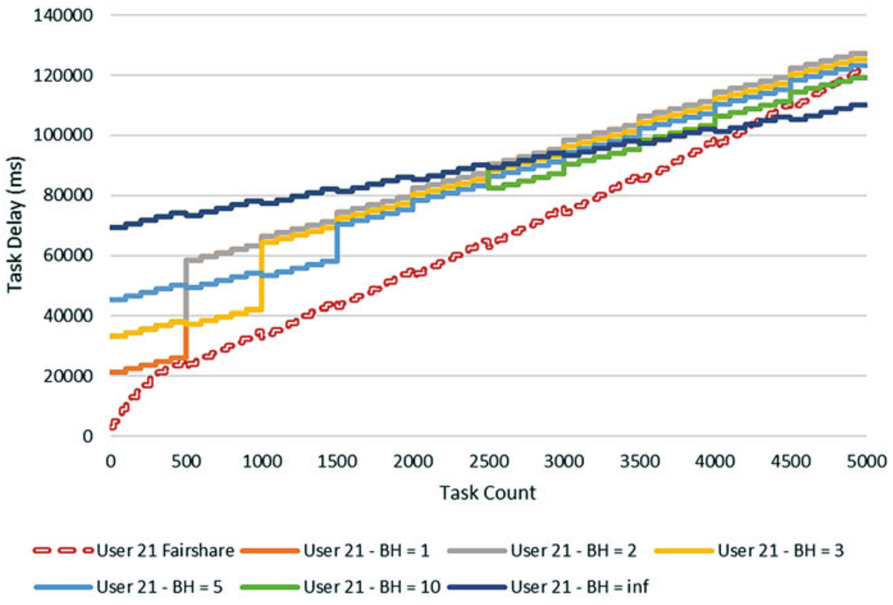


**Fig. 10.13** User 21's task delays with all BH values in simulation 14

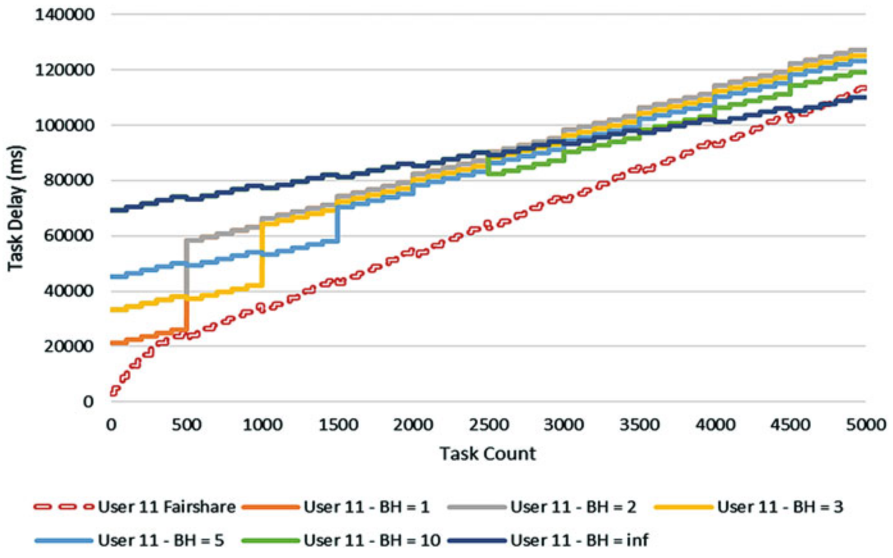


**Fig. 10.14** User 11's task delays with all BH values in simulation 10

10, the smaller users (1–10) saw an increase in performance. All of these users saw the most improvement with BH = *inf* (Table 10.1), at which users 1–10 saw an average increase in performance greater than 95%.

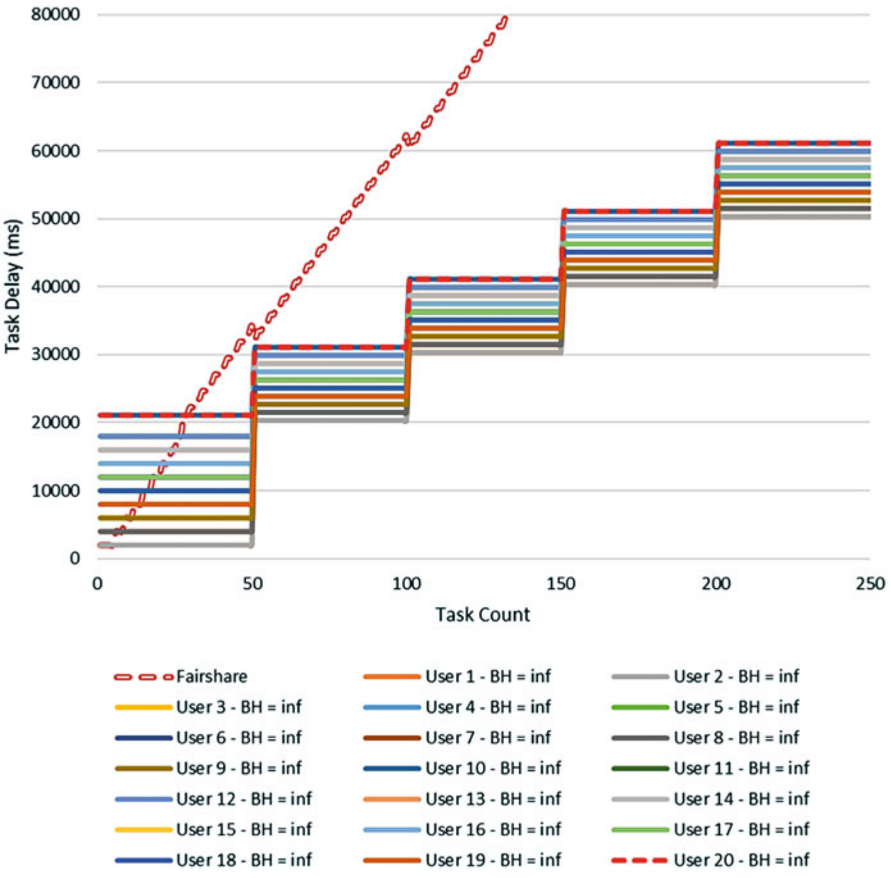


**Fig. 10.15** The task delays of users 1–20 with BH = *inf* in simulation 14

**Table 10.4** Comparison of the delays experienced by user 1 and user 11 under Fair-Share (FS) and Rawlsian Fair (RF) in simulation 10

| | FS delay (ms) | RF delay (ms) BH = 1 | RF delay (ms) BH = 2 | RF delay (ms) BH = 3 | RF delay (ms) BH = 5 | RF delay (ms) BH = 10 | RF delay (ms) BH = inf |
|---|---|---|---|---|---|---|---|
| User 1 | 67,133 | 21,361 | 26,421 | 30,021 | 34,121 | 28,641 | 28,641 |
| vs Fairshare | N/A | 214% | 154% | 124% | 97% | 133% | 133% |
| User 11 | 63,302 | 85,891 | 87,091 | 83,381 | 83,271 | 90,251 | 89,701 |
| vs Fairshare | N/A | −26% | −27% | −24% | −24% | −30% | −29% |

**Table 10.5** Comparison of the delays experienced by user 1 and user 21 under Fair-Share (FS) and Rawlsian Fair (RF) in simulation 14

| | FS delay (ms) | RF delay (ms) BH = 1 | RF delay (ms) BH = 2 | RF delay (ms) BH = 3 | RF delay (ms) BH = 5 | RF delay (ms) BH = 10 | RF delay (ms) BH = inf |
|---|---|---|---|---|---|---|---|
| User 1 | 73,617 | 20,401 | 25,941 | 29,541 | 33,881 | 28,641 | 28,641 |
| vs Fairshare | N/A | 261% | 184% | 149% | 117% | 157% | 157% |
| User 21 | 65,245 | 85,891 | 87,091 | 83,381 | 83,271 | 90,251 | 89,701 |
| vs Fairshare | N/A | −24% | −25% | −22% | −22% | −28% | −27% |

Table 10.5 compares the delay of a smaller user (user 1) in simulation 14 to that of the larger user (user 21). As expected, user 1 saw an increase in performance with every BH value, and all users saw performance increases with BH = *inf* (Table 10.1). The average improvement in performance was over 110%.

# Chapter 11
# Class B Results and Analysis

## 11.1 Class B Simulations

Class B simulations describe situations in which one user submits an intermittent workload while another user or other users submits a steady workload. In simulation 2, two users submit workloads that are steady and equal in the number of tasks, while one user submits an intermittent workload. As the number of users increases—to 6 in simulation 7, 11 in simulation 11, and 21 in simulation 15—one user's workload remains constant at an intermittent rate of 200 tasks for every two PCs up to 5000 tasks.

As the number of users increases, the steady workload is distributed among the users. In simulation 7, users 1–5 submit 20 tasks/PC each, for a total of 5000 tasks, while user 6 submits 5000 tasks on their own. In simulation 11, users 1–10 submit 10 tasks/PC each, while user 11 submits 5000 tasks on their own. In simulation 15, users 1–20 submit 20 tasks/PC each, for a total of 5000 tasks, while user 21 submits 5000 tasks intermittently.

## 11.2 Class B Results

When the workloads are equal, as in simulation 3, a difference in performance of less than 2% is observed between fair-share and Rawlsian Fair. As the gap between the largest user and the smallest user increases, however, Rawlsian Fair demonstrates a performance that is up to 200% better than that of Fair-Share (Table 11.1). The drop in performance improvement can be explained by the fact that while all 20 users submit at a rate of 10 tasks/PC, for a total of 200 tasks, the total number of resources remains at 100. Even though the scheduler dedicates all of the resources to users 1–20, the length of the delay is double that observed in other tests simply because

**Table 11.1** Class B Submission type average delay

| Class B Average delay | | | | |
|---|---|---|---|---|
| | Simulation 3 | Simulation 7 | Simulation 11 | Simulation 15 |
| User under test | User 1 | Users 1–5 | Users 1–10 | Users 1–20 |
| Fairshare average delay (ms) | 59,698 | 54,927 | 66,144 | 61,113 |
| Average Rawlsian Fair delay (ms) | 54,537 | 21,873 | 21,873 | 34,425 |
| Rawlsian Fair delay per user (ms) | 54,537 | User 1: 21,801 | User 1: 21,083 | User 1: 32,049 |
| BH = *inf* | | User 2: 21,803 | User 2: 21,083 | User 2: 32,049 |
| | | User 3: 21,783 | User 3: 22,663 | User 3: 33,585 |
| | | User 4: 22,663 | User 4: 22,663 | User 4: 33,585 |
| | | User 5: 22,663 | User 5: 22,663 | User 5: 33,585 |
| | | | User 6: 21,08 | User 6: 35,297 |
| | | | User 7: 21,083 | User 7: 32,049 |
| | | | User 8: 22,663 | User 8: 32,049 |
| | | | User 9: 22,663 | User 9: 32,049 |
| | | | User 10: 21,083 | User 10: 36,769 |
| | | | | User 11: 36,769 |
| | | | | User 12: 36,769 |
| | | | | User 13: 36,769 |
| | | | | User 14: 35,297 |
| | | | | User 15: 35,297 |
| | | | | User 16: 35,297 |
| | | | | User 17: 35,297 |
| | | | | User 18: 33,585 |
| | | | | User 19: 33,585 |
| | | | | User 20: 36,769 |
| Average improvement | <2% | 156% | 203% | 78% |
| Improvement per user | <2% | User 1:165% | User 1: 214% | User 1 91% |
| | | User 2: 165% | User 2: 214% | User 2 91% |
| | | User 3: 155% | User 3: 192% | User 3 82% |
| | | User 4: 147% | User 4: 192% | User 4 82% |
| | | User 5: 147% | User 5: 192% | User 5 82% |
| | | | User 6: 214% | User 6 73% |
| | | | User 7: 214% | User 7 91% |
| | | | User 8: 192% | User 8 91% |
| | | | User 9: 192% | User 9 91% |
| | | | User 10: 214% | User 1 66% |
| | | | | User 11: 66% |
| | | | | User 12: 66% |
| | | | | User 13: 66% |
| | | | | User 14: 73% |
| | | | | User 15: 73% |
| | | | | User 16: 73% |
| | | | | User 17: 73% |
| | | | | User 18: 82% |
| | | | | User 19: 82% |
| | | | | User 20: 66% |

there are not enough resources available at any given time. In simulation 11, the users submit at a rate of 10 tasks/PC, but there are only 10 users. Similarly, in simulation 7, 5 users submit at a rate of 20 tasks/PC.

### *11.2.1 Simulation 3 Analysis and Results*

In simulation 3, one user sent a continuous load and another user sent a load in start–stop fashion. Within a given span of two PCs, both users sent the same number of tasks. User 1 sent 100 tasks/PC and user 2 sent 200 tasks every two PCs. Both sent 5000 tasks in total (Fig. 11.1). As expected, no dramatic differences in performance were observed in this two-user simulation (Table 11.2).

Simulation 3 featured 2 users and was run for all BH options and fair-share. As expected, the results were similar for Rawlsian Fair and Fair-Share. Individual results are presented to depict the cases with practically identical submission profiles. Table 11.2 outlines the performance characteristics of the two users in simulation 2. Figures 11.2 and 11.3 show the variance in performance between Fair-Share and Rawlsian Fair with BH = 1. The variance is minimal for the first 1000 tasks, and then it aligns nicely with fair-share. The same pattern is observed to a greater extent with BH = *inf,* as is shown in Figs. 11.4 and 11.5, in which the task delays for fair-share and BH = *inf* practically match.

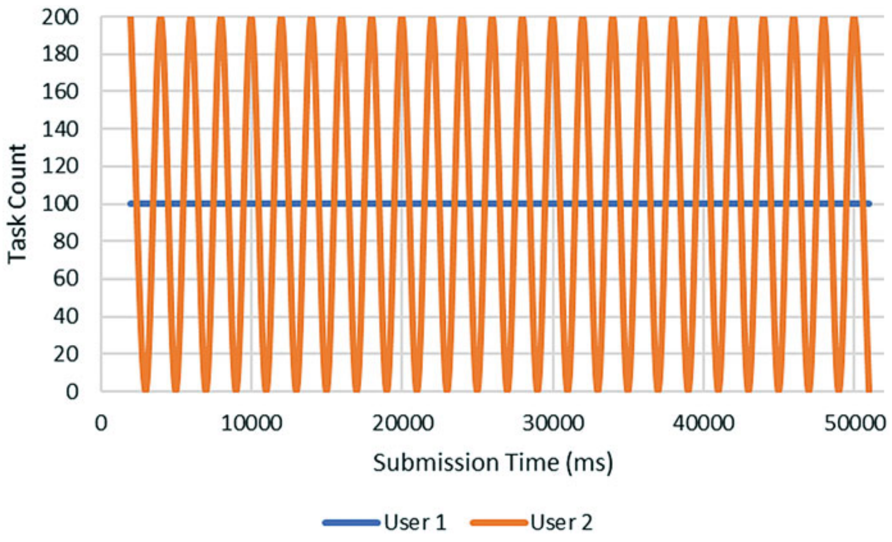


**Fig. 11.1** Task submission in simulation 3 by user 1 and user 2

**Table 11.2** Comparison of the Fair-Share (FS) and Rawlsian Fair (RF) delays in simulation 3

| | FS delay (ms) | RF delay (ms) BH = 1 | RF delay (ms) BH = 2 | RF delay (ms) BH = 3 | RF delay (ms) BH = 5 | RF delay (ms) BH = 10 | RF delay (ms) BH = inf |
|---|---|---|---|---|---|---|---|
| User 1 | 55,698 | 53,301 | 60,561 | 51,229 | 49,349 | 73,337 | 54,537 |
| vs Fairshare | N/A | 4.50% | −8.03% | 8.72% | 12.87% | −24.05% | 2.13% |
| User 2 | 54,288 | 58,345 | 51,025 | 60,337 | 62,137 | 37,929 | 55,037 |
| vs Fairshare | N/A | −6.95% | 6.39% | −10.03% | −12.63% | 43.13% | −1.36% |

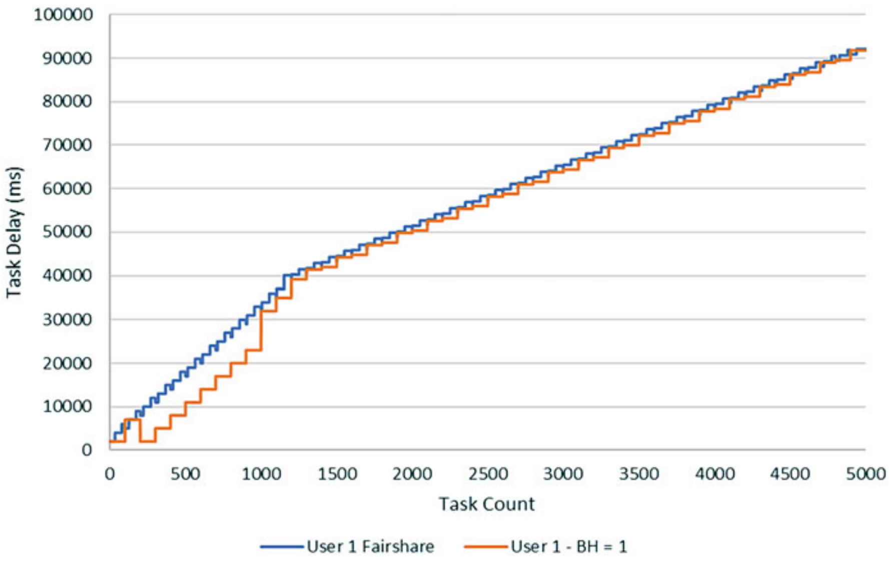


**Fig. 11.2** User 1's task delay with Fairshare and BH = 1 in simulation 3

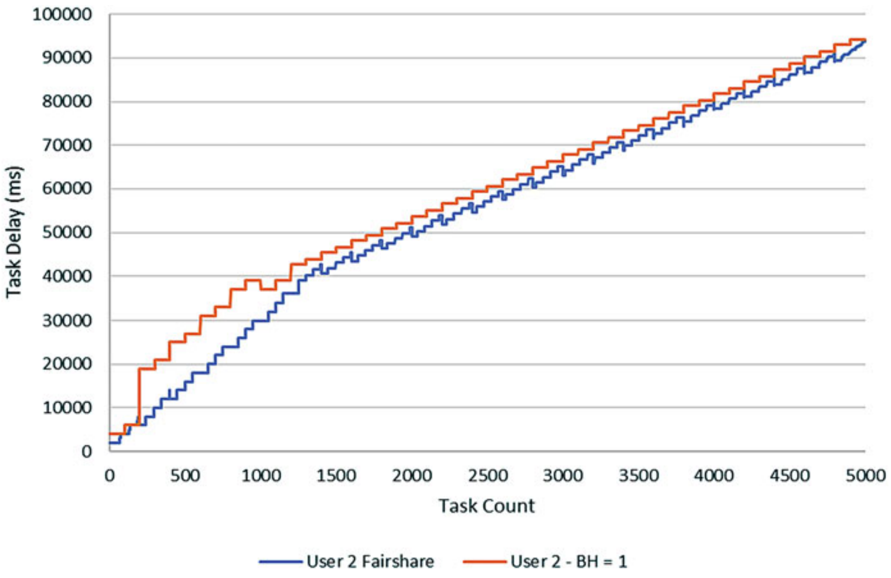


**Fig. 11.3** User 2's task delay with Fairshare and BH = 1 in simulation 3

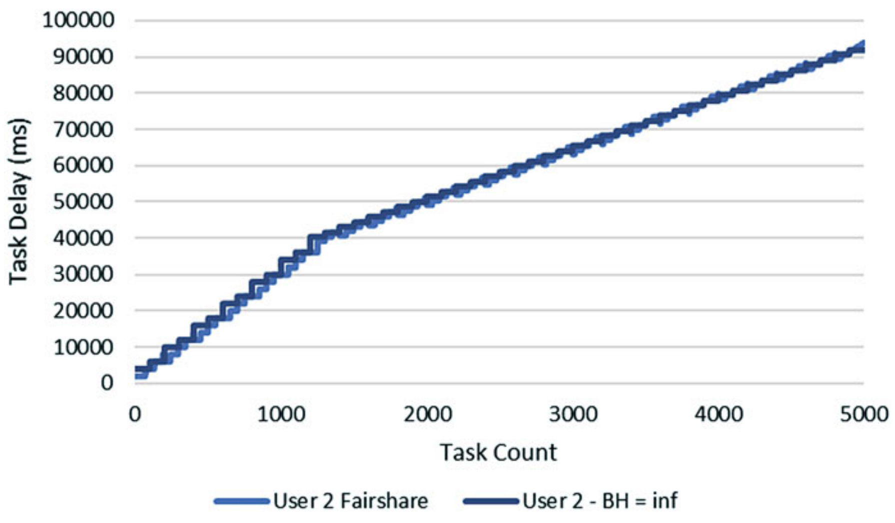


**Fig. 11.4** User 2's task delay—comparing Fair-Share with BH = *inf* in simulation 3

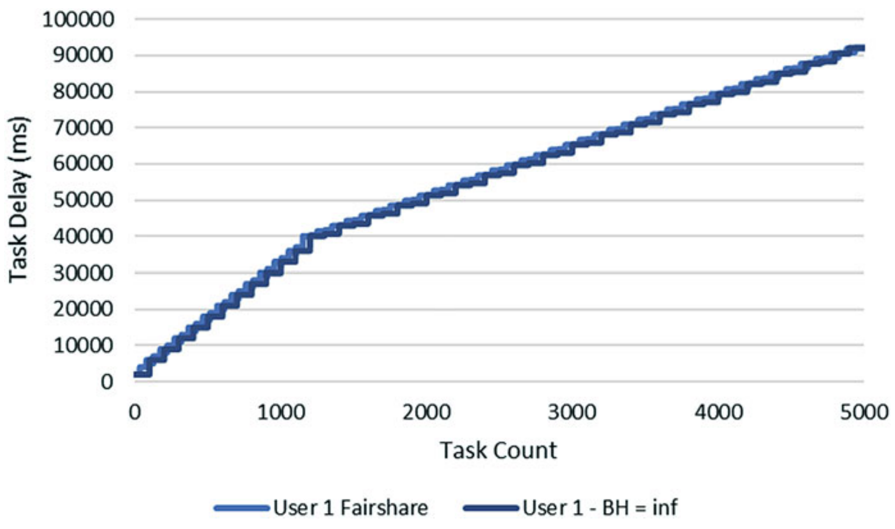


**Fig. 11.5** User 1's task delay—comparing Fair-Share with BH = *inf* in simulation 3

### *11.2.2 Simulation 7 Analysis and Results*

In simulation 7, one user submitted 5000 tasks, and five other users submitted 1000 tasks each (Fig. 11.6). The difference between simulation 7 and simulation 3 is that in simulation 7, users 1–5 submitted continuously and simultaneously at a rate of 20 tasks/PC.

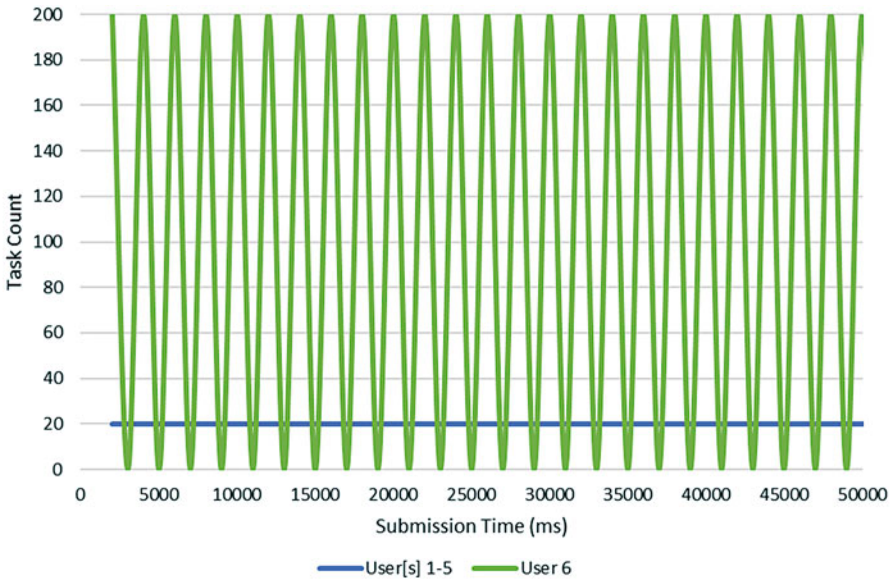


**Fig. 11.6** The task submissions of all 6 users in simulation 7

**Table 11.3** Comparison of the delays generated by Fair-Share (FS) and Rawlsian Fair (RF) in simulation 7

| | FS delay (ms) | RF delay (ms) BH = 1 | RF delay (ms) BH = 2 | RF delay (ms) BH = 3 | RF delay (ms) BH = 5 | RF delay (ms) BH = 10 | RF delay (ms) BH = inf |
|---|---|---|---|---|---|---|---|
| User 1 | 55,866 | 24,349 | 25,561 | 24,981 | 24,261 | 22,153 | 21,083 |
| vs Fairshare | N/A | 129% | 119% | 124% | 130% | 152% | 165% |
| User 6 | 54,255 | 85,721 | 82,865 | 83,025 | 80,713 | 75,601 | 87,701 |
| vs Fairshare | N/A | −37% | −35% | −35% | −33% | −28% | −38% |

The remainder of this section compares Rawsian Fair scheduling to Fair-Share for users 1–5, and again for user 6. The other comparison is amongst users 1–5 for a given simulation parameter. Table 11.3 shows the performance characteristics of user 1 compared to those of the larger user 6.

Figure 11.7 shows the task delays that user 1 experienced with the different BH values. User 1 submitted their tasks first, followed by user 2, and so on. As previously mentioned, the FCFS aspect of the Rawlsian Fair scheduler delays the execution of tasks from some users with equal seniority. All of the users were better off than they would have been under Fair-Share, but user 1’s tasks were completed first (Fig. 11.8). The performance level that user 1 experienced under Rawlsian Fair was as much as 165% better than the performance level they experienced under fair-

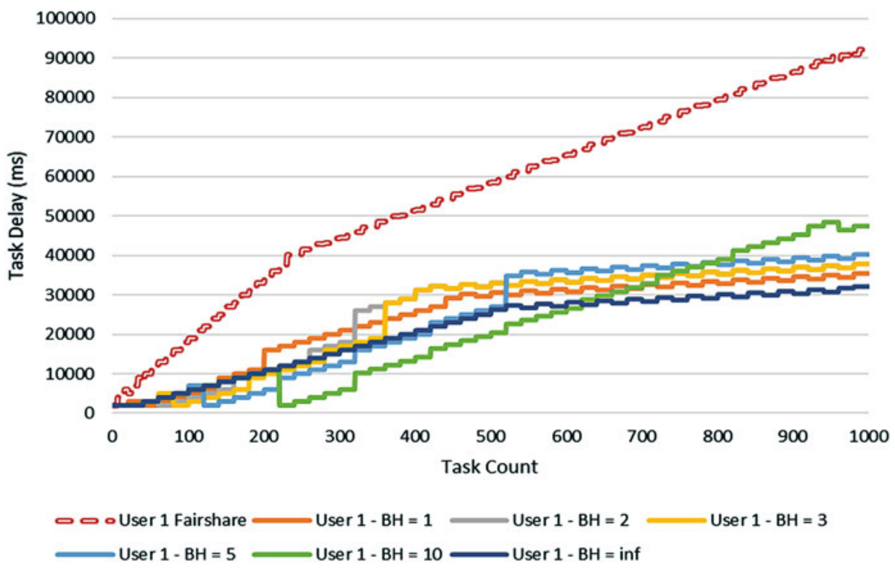


**Fig. 11.7** User 1's task delay with Fair-Share and all BH values in simulation 7

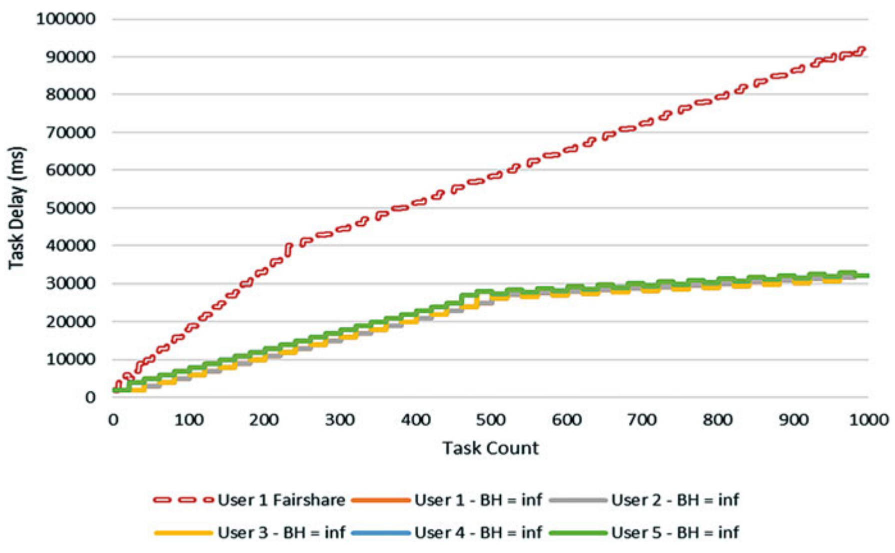


**Fig. 11.8** The task delays experienced by users 1–5 with BH = *inf* and Fair-Share in simulation 7

share. Similarly, the performance levels that users 2–5 experienced under Rawlsian Fair were approximately 140%–165% better than the performance levels they experienced under Fair-Share (Table 11.1). However, the performance level that user 6 experienced under Rawlsian Fair was approximately 35% lower than the performance level they experienced under Fair-Share. Rawlsian Fair pushed out user

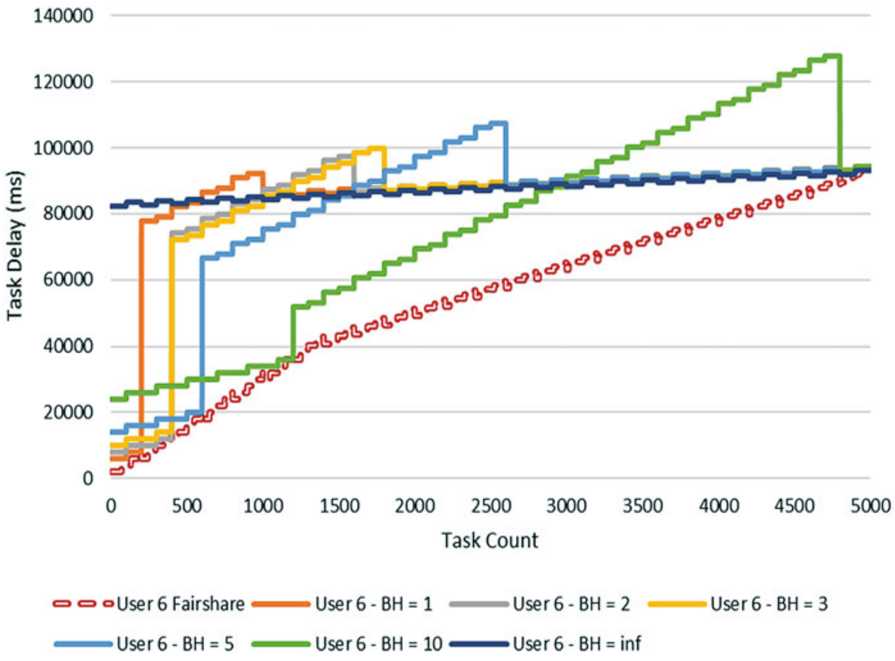


**Fig. 11.9** User 6's task delay with all BH values and Fair-Share in simulation 7

6 in favor of the smaller users (users 1–5). This is shown in Fig. 11.9. The dotted red line traversing the graph represents the delay caused by the fair-share scheduling algorithm.

The rise and the fall of the task delay with various bucket-history (BH) values resulted from abrupt changes in the seniorities of tasks for the smaller user compared with the larger user. This feature is further zoomed in Fig. 11.10, in which an abrupt increase in the task delay is a product of the BH values and the submission profile for user 6 (this is marked with a red circle). User 6 submitted 200 tasks for every 2 PC ticks, starting at PC = 2. With BH = 1, the rise occurs after 200 tasks, or a single bucket worth of tasks compared to other users. With BH = 2, the rise occurs after 400 tasks: 200 tasks submitted at PC = 2 and 200 tasks submitted at PC = 4. With BH = 3, the rise is the same at 400 tasks because user 6 submits tasks every other PC. This trend continues with rises at 600 tasks with BH = 5 and at 1200 tasks with BH = 10. BH = *inf* is not represented in this figure because it normalizes the submission across all PC ticks.

The effect of the BH values on FUD parameters is apparent in this simulation: dynamicity was higher with lower BH values, and it was lower with higher BH values. At BH = *inf*, the delay was barely affected by the submission pattern of user 6 and was only affected by the overall difference in the task-submission count.

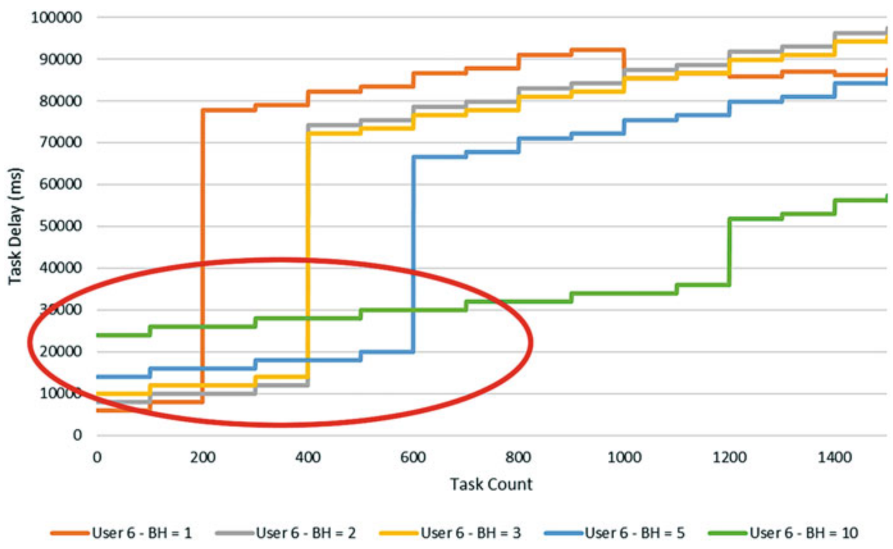


**Fig. 11.10** A close-up of user 6's (i.e. the largest user's) task delay under Rawlsian Fair with different BH values in simulation 7

### *11.2.3 Simulations 11 and 15 Results and Analysis*

Simulation 11 featured 11 users, and simulation 15 featured 21 users. The total number of tasks submitted in both simulations was the same, but the number of tasks submitted by each user decreased as the number of users increased. In simulation 11, 10 small users each submitted 10 tasks/PC up to a total of 500 tasks (5000 for all 10 users). User 11 submitted 500 tasks every two PCs up to a total of 5000 tasks (Fig. 11.11). In simulation 15, 20 small users each submitted 10 tasks/PC up to a total of 250 tasks (5000 for all 20 users). User 21 submitted 200 tasks for every two PCs up to a total of 5000 tasks (Fig. 11.12).

In simulation 11, the BH value had very little effect on the execution of the small users' tasks. With all BH values, the smaller users' tasks were completed more quickly than they were with fair-share (Fig. 11.13). The BH value did affect the execution of the larger user's tasks, however: tasks submitted earlier incurred shorter delays than did tasks submitted later depending on the BH value. With BH = 3, for example, the larger user's first 1000 tasks were completed more quickly than were their other 4000 tasks. This was because their first 1000 tasks were submitted within the first 3 PC values (500 at PC = 2 and 500 at PC = 4). With BH = 3, until there were at least 3 buckets, the bucket history did not affect the smaller users. Once three buckets had been created, however, proper Rawlsian Fair scheduling took place. Similarly, in simulation 15, the BH value did not affect the scheduler's behavior toward the small users (Fig. 11.14). The tasks submitted by the larger user (user 21) were delayed until the smaller users' tasks had finished, but as in simulation 11 (Fig. 11.15), the effect of Rawlsian Fair scheduling vis-à-vis apparent seniority

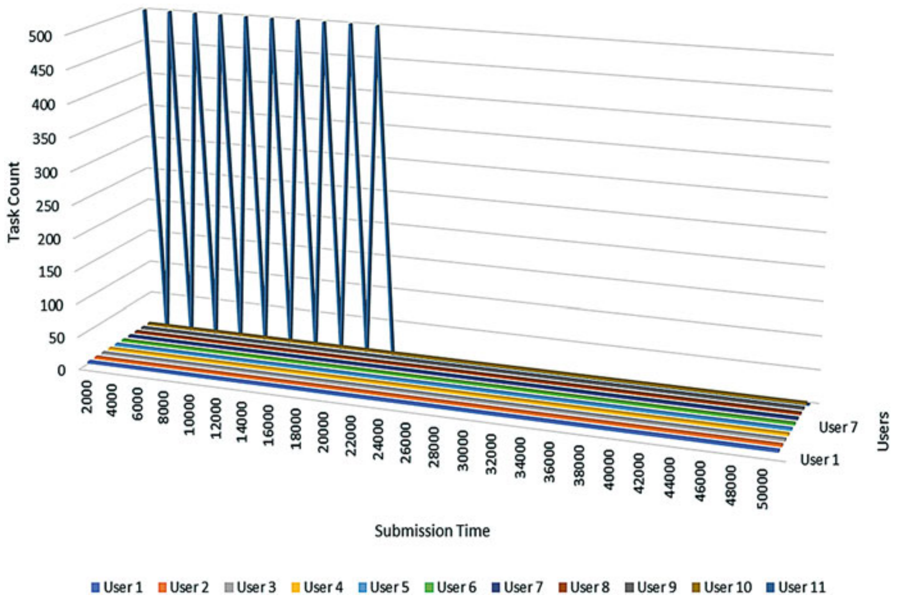


**Fig. 11.11** The task submissions of all 11 users in simulation 11

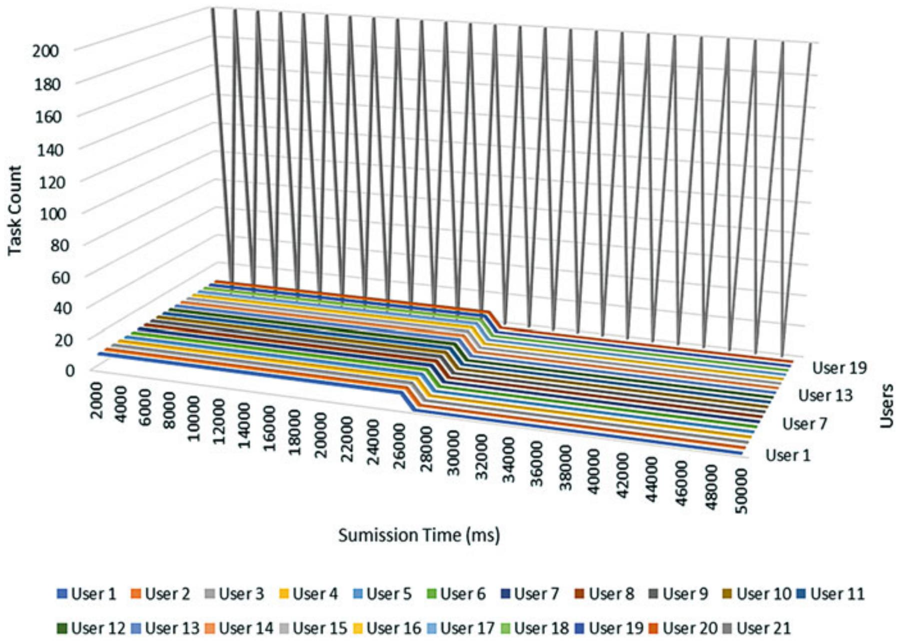


**Fig. 11.12** The task submissions of all 21 users in simulation 15

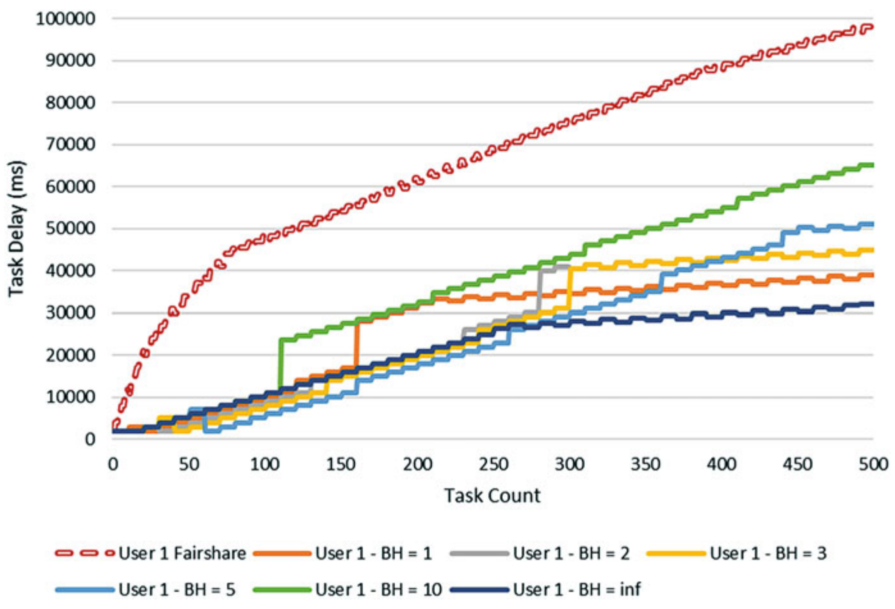


**Fig. 11.13** User 1's task delays with all BH values in simulation 11

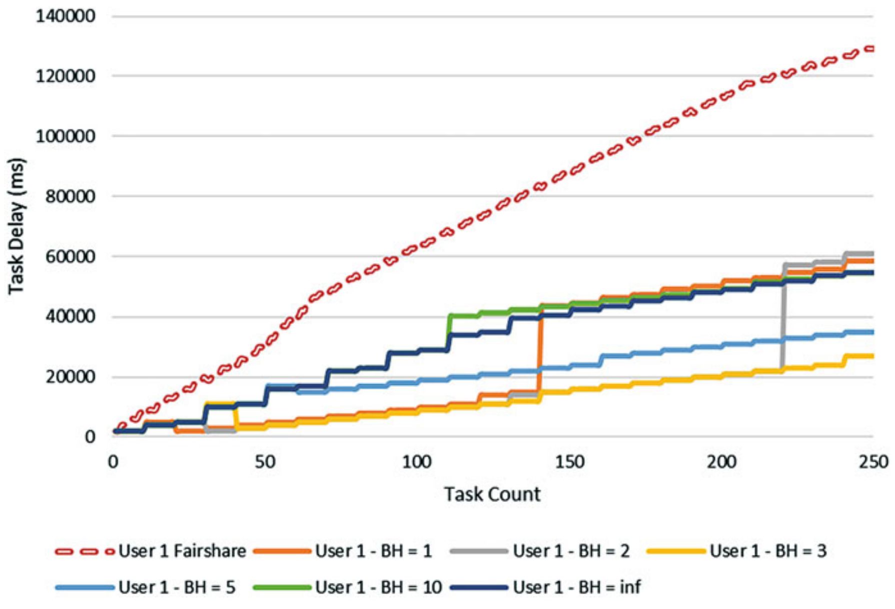


**Fig. 11.14** User 1's task delays with all BH values in simulation 15

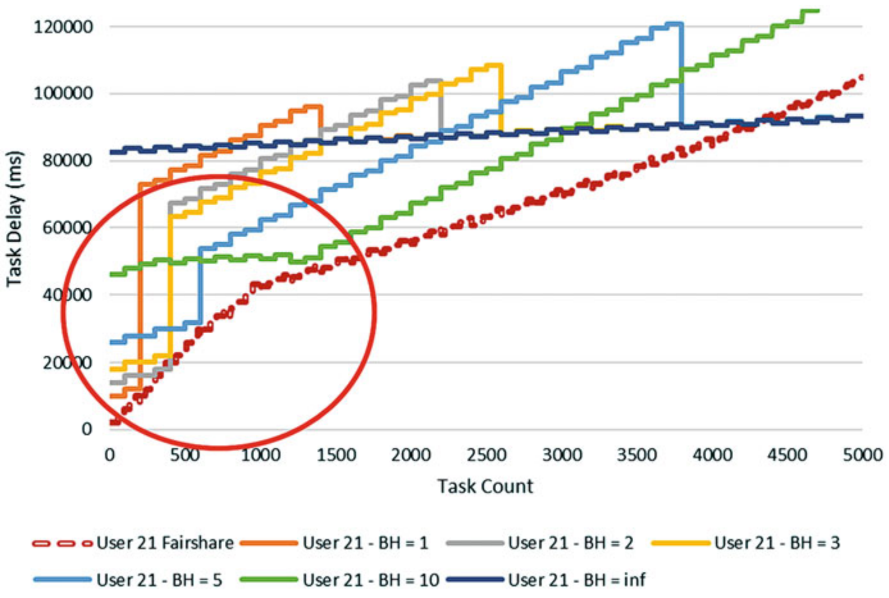


**Fig. 11.15** User 21's task delays with all BH values in simulation 1

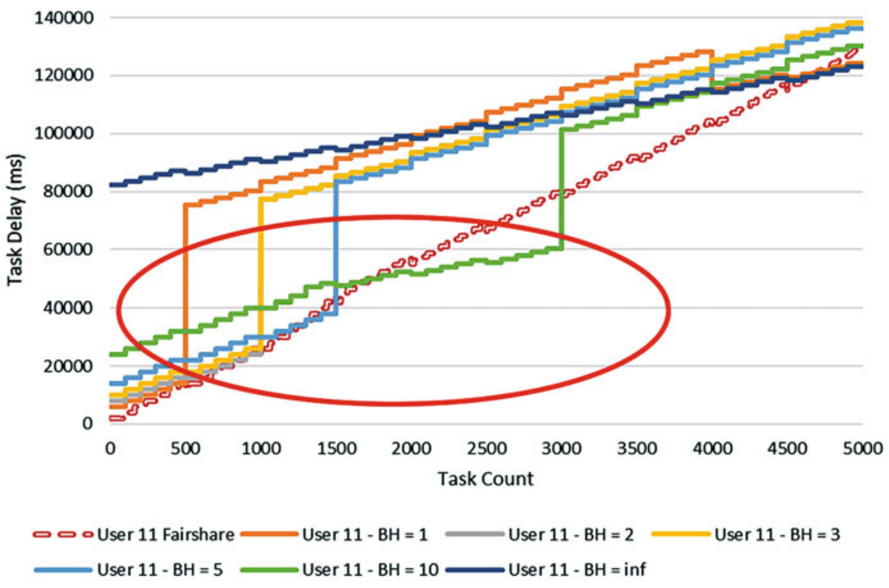


**Fig. 11.16** User 11's task delays with all BH values in simulation 11

was based on the BH values (Figs. 11.16 and 11.17). Overall, the smaller users' tasks were completed more quickly under Rawlsian Fair scheduling than under fair-share (Fig. 11.18).

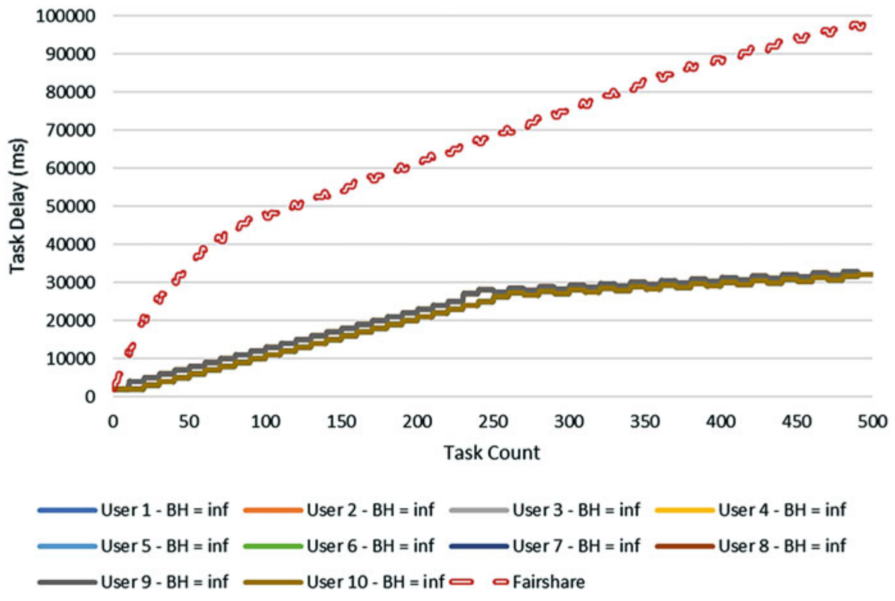


**Fig. 11.17** The task delays experienced by users 1–10 with BH = *inf* in simulation 11

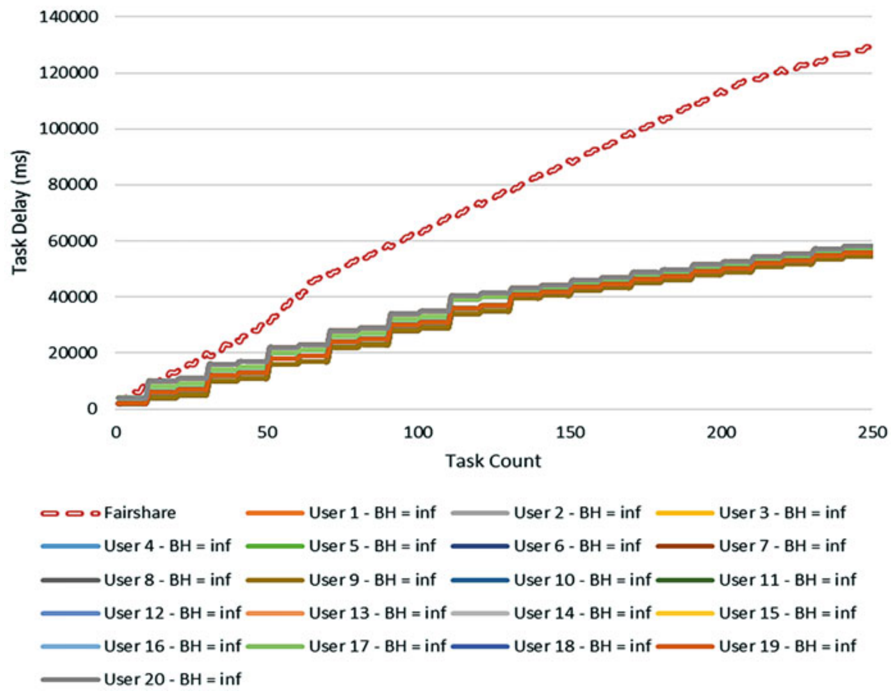


**Fig. 11.18** The task delays experienced by users 1–20 with BH = *inf* in simulation 15

**Table 11.4** Comparison of the delays experienced by user 1 and user 11 under Fair-Share (FS) and Rawlsian Fair (RF) in simulation 11

| | FS delay (ms) | RF delay (ms) BH = 1 | RF delay (ms) BH = 2 | RF delay (ms) BH = 3 | RF delay (ms) BH = 5 | RF delay (ms) BH = 10 | RF delay (ms) BH = inf |
|---|---|---|---|---|---|---|---|
| User 1 | 66,199 | 26,421 | 26,641 | 25,741 | 24,405 | 35,937 | 21,083 |
| vs Fairshare | N/A | 151% | 148% | 157% | 171% | 84% | 214% |
| User 11 | 66,186 | 96,221 | 89,441 | 89,841 | 84,661 | 73,319 | 102,701 |
| vs Fairshare | N/A | −31% | −26% | −26% | −22% | −10% | −36% |

**Table 11.5** Comparison of the delays experienced by user 1 and user 21 under Fair-Share (FS) and Rawlsian Fair (RF) in simulation 15

| | FS delay (ms) | RF delay (ms) BH = 1 | RF delay (ms) BH = 2 | RF delay (ms) BH = 3 | RF delay (ms) BH = 5 | RF delay (ms) BH = 10 | RF delay (ms) BH = inf |
|---|---|---|---|---|---|---|---|
| User 1 | 61,113 | 26,221 | 16,565 | 12,761 | 20,921 | 33,081 | 32,049 |
| vs Fairshare | N/A | 133% | 269% | 379% | 192% | 85% | 91% |
| User 21 | 61,784 | 85,017 | 82,517 | 82,837 | 81,361 | 80,477 | 87,801 |
| vs Fairshare | N/A | −27% | −25% | −25% | −24% | −23% | −30% |

Table 11.4 compares the performance characteristics of one of the smaller users (user 1) to those of the larger user (user 11). With all BH values in simulation 11, the smaller users (users 1–10) saw in an increase in performance. All of the users saw the most improvement with BH = *inf* (Table 11.1), with which the average performance for users 1–10 increased by over 200%.

Table 11.5 compares a smaller user (user 1) in simulation 15 with the larger user (user 21). As expected, user 1 saw an increase in performance with every BH value, and all users saw performance increases with BH = *inf* (Table 11.1), which generated an average performance increase of about 80%.

# Chapter 12
# Class C Results and Analysis

## 12.1 Class C Simulations

Class C simulations describe cases featuring a small user and one or more disproportionately large users. In Simulation 4, one user submits a large workload (5000), and another user submits a 50-task, one-time workload at a later time. As the number of users increases—to 6 in Simulation 8, 11 in Simulation 12, and 21 in Simulation 16—the smaller user's workload remains constant while the larger users' workloads are distributed to one or more other users. In Simulation 8, users 1–5 submit 1000 tasks each and 5000 together. In Simulation 12, users 1–10 submit 500 tasks each, and user 11 submits 5000 tasks on their own. In Simulation 16, users 1–20 submit 250 tasks each.

## 12.2 Class C Results

As Table 12.1 shows, all class C workloads benefit from Rawlsian Fair scheduling. Each of the Class C simulations that used Rawlsian Fair performed 9 times (~900%) better than did those that used fair-share scheduling.

### *12.2.1 Simulation 4 Analysis and Results*

Simulation 4 was the first scenario in which the two users did not send the same total number of tasks. It was also the first simulation in which Rawlsian Fair performed dramatically better (close to 900% better) than did traditional fair-share (Table 12.2). The tasks were intentionally submitted at a time where the effects of varying of the Bucket History (BH) sizes can be realized vis-à-vis the submission pattern.

A. Sedighi, M. Smith, *Fair Scheduling in High Performance Computing Environments*, https://doi.org/10.1007/978-3-030-14568-2_12

**Table 12.1** Class C Submission type average delay

| Class C Average delay (BH = *inf*) | | | | |
|---|---|---|---|---|
| | Simulation 4 | Simulation 8 | Simulation 12 | Simulation 16 |
| User under test | User 2 | User 6 | User 11 | User 21 |
| Fairshare delay (ms) | 32,317 | 32,917 | 33,037 | 33,189 |
| Rawlsian Fair delay (ms) | 3301 | 3301 | 3301 | 3301 |
| Improvement | 879%<br>~9x | 897%<br>~9x | 901%<br>~9x | 905%<br>~9x |

**Table 12.2** Comparison of the delays generated by fair-share (FS) and Rawlsian Fair (RF) in simulation 4

| | FS delay (ms) | RF delay (ms) BH = 1 | RF delay (ms) BH = 2 | RF delay (ms) BH = 3 | RF delay (ms) BH = 5 | RF delay (ms) BH = 10 | RF delay (ms) BH = inf |
|---|---|---|---|---|---|---|---|
| User 1 | 65,501 | 64,301 | 64,301 | 64,301 | 64,301 | 65,501 | 65,501 |
| vs Fairshare | N/A | 1.87% | 1.87% | 1.87% | 1.87% | 0.00% | 0.00% |
| User 2 | 32,317 | 59,601 | 59,601 | 59,601 | 59,601 | 3301 | 3301 |
| vs Fairshare | N/A | −45.78% | −45.78% | −45.78% | −45.78% | 879.01% | 879.01% |

Table 12.2 compares the performance characteristics of Rawlsian Fair and fair-share in simulation 4.

In simulation 4, user 1 sent a single batch of 5000 tasks, which were sorted into a single bucket. User 2 submitted a one-time load of 50 tasks at PC = 7. As a result, the first bucket held user 1's 5000 tasks, and all of the buckets remained empty until bucket 7, which held user 2's 50 tasks. PC = 7 was chosen to demonstrate how the algorithm behaves with different BH values. At low BH values—e.g. 1, 2, 3, and 5—the First-Come-First-Served feature of the Rawlsian Fair scheduler kicked in, executing user 1's tasks ahead of user 2's. At BH values of 10 and *inf*, the scheduler is capable of picking user 2's tasks, and schedule them for execution (Fig. 12.2). The red line traversing the figure represents the delay incurred by user 2 under fair-share. In contrast, user 1's tasks were completed as in the Fair-Share scenario (Fig. 12.1); user 1's total delay was unaffected by the scheduling decisions made by the Rawlsian Fair scheduler.

## 12.2.2 Simulation 8 Results and Analysis

In Simulation 8, users 1–5 sent 1000 tasks each in one-time bursts at the beginning of the simulation, submitting a total of 5000 tasks. User 6 sent a one-time load of 50 tasks at PC = 7. Because all of the tasks for users 1–5 were submitted in the first bucket (5000 tasks in bucket 2), the delay incurred under various BH values was

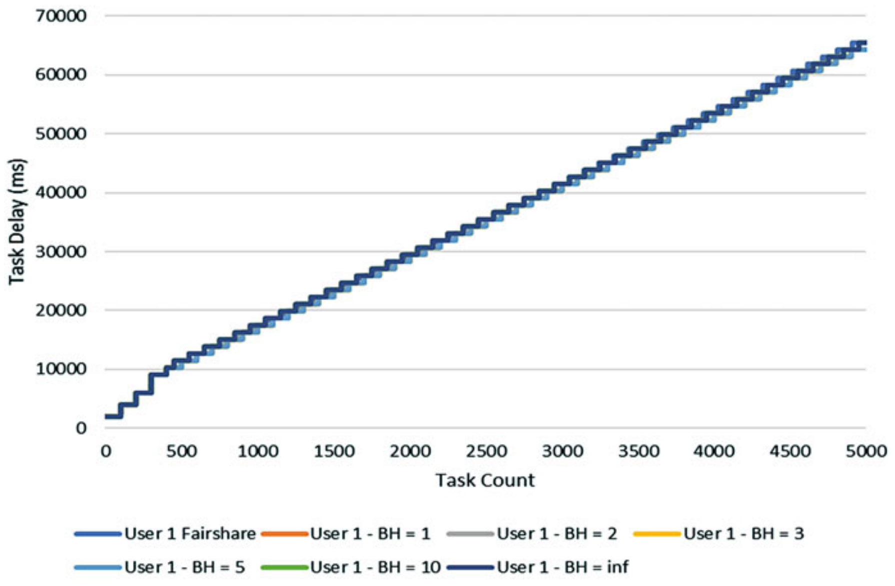


**Fig. 12.1** User 1's task delays with fair-share and all BH values in simulation 4

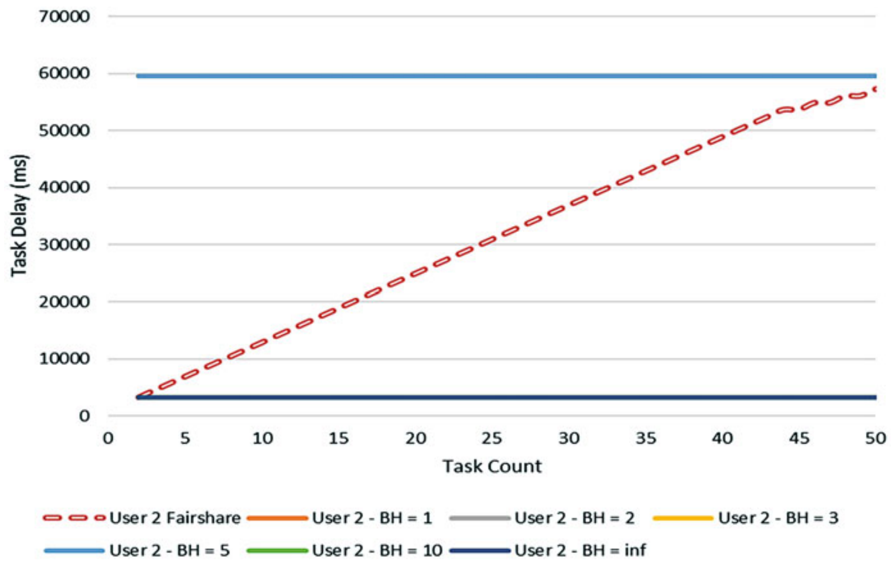


**Fig. 12.2** User 2's task delays with fair-share and all BH values in simulation 4

virtually the same for a single user (Fig. 12.3). With smaller BH values—such as 1, 2, 3, and 5—the FCFS feature of the Rawlsian Fair scheduler executed all of the tasks for a given user at once before moving on to the next user, which included User

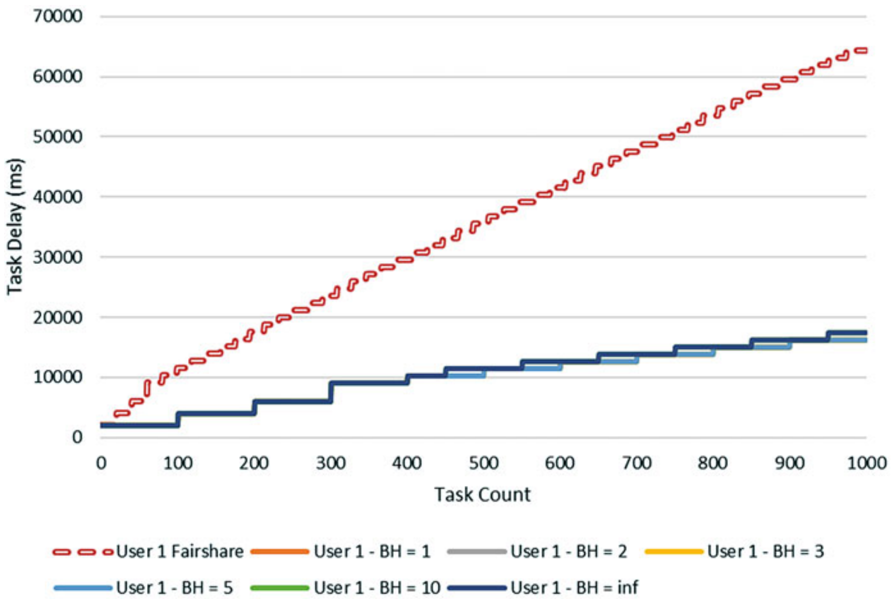


**Fig. 12.3** Users 1's task delay with all BH values in simulation 8

**Table 12.3** Comparison of the delays generated by Fair-Share (FS) and Rawlsian Fair (RF) in simulation 8

| | FS delay (ms) | RF delay (ms) BH = 1 | RF delay (ms) BH = 2 | RF delay (ms) BH = 3 | RF delay (ms) BH = 5 | RF delay (ms) BH = 10 | RF delay (ms) BH = inf |
|---|---|---|---|---|---|---|---|
| User 6 | 32,918 | 59,601 | 59,601 | 59,601 | 59,601 | 3301 | 3301 |
| vs Fairshare | N/A | −45% | −45% | −45% | −45% | 897% | 897% |

6. This is shown for users 1–5 in Fig. 12.5. The dotted red line in the figure indicates the delays generated by the fair-share scheduling algorithm. Table 12.3 shows the difference in performance level experienced by user 6: Rawlsian Fair with BH = *inf* performed about 900% better than did Fair-Share.

User 6's 50 tasks were submitted into bucket 7 after users 1–5 had submitted their tasks. As a result, user 6's tasks were pending behind those of users 1–5. With BH values of 1, 2, 3, and 5, user 6's tasks were executed on a first-come-first-served basis. With BH values of 10 and *inf* (Fig. 12.4), however, they were scheduled for execution immediately because of their seniority. Because the tasks submitted by users 1–5 were executed in the order in which they arrived, user 1 saw an increase in performance, while user 5 saw a drop in performance of close to 40% (Table 12.4). The first-come-first-served aspect of the Rawlsian Fair scheduler can be gamed when users have *identical* workloads.

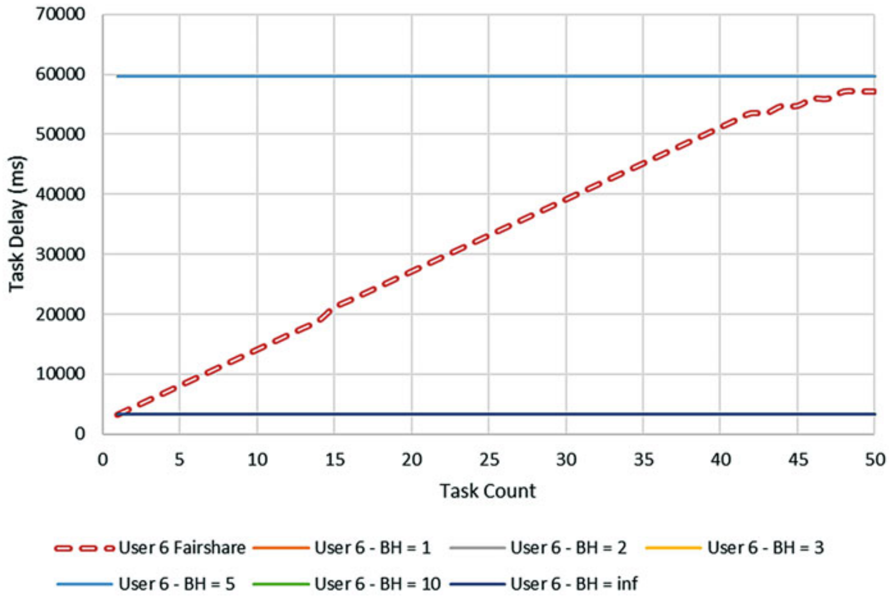


**Fig. 12.4** User 6's task delay with all BH values in simulation 8

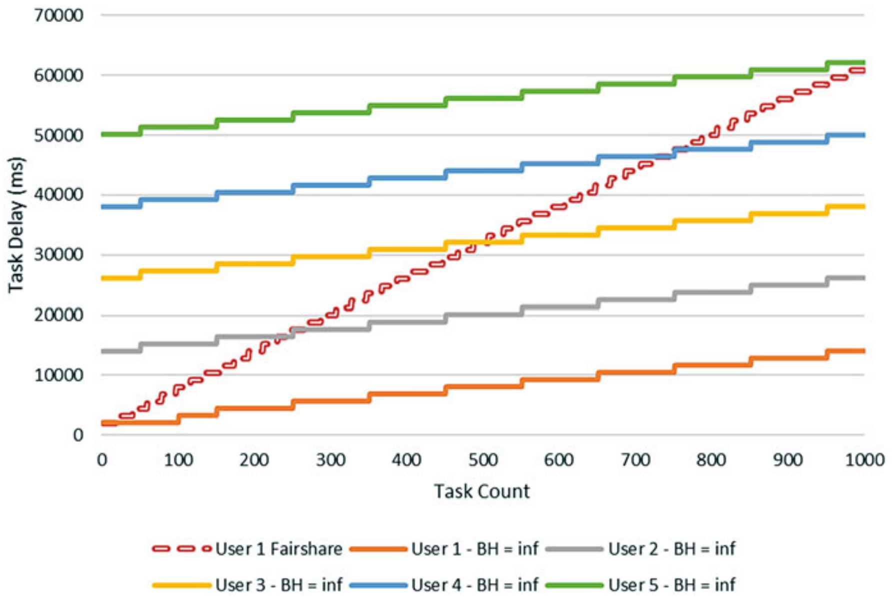


**Fig. 12.5** The task delays experienced by users 1–5 with BH = *inf* in simulation 8

**Table 12.4** Comparison of the delays experienced by users 1–5 with Fair-Share (FS) and Rawlsian Fair (RF) with BH = *inf* in simulation 8

| | FS delay (ms) | User 1 RF delay (ms) | User 2 RF delay (ms) | User 3 RF delay (ms) | User 4 RF delay (ms) | User 5 RF delay (ms) |
|---|---|---|---|---|---|---|
| BH = inf | 32,174 | 8041 | 20,101 | 32,101 | 44,101 | 56,101 |
| vs Fairshare | N/A | 340% | 76% | 10% | −20% | −37% |

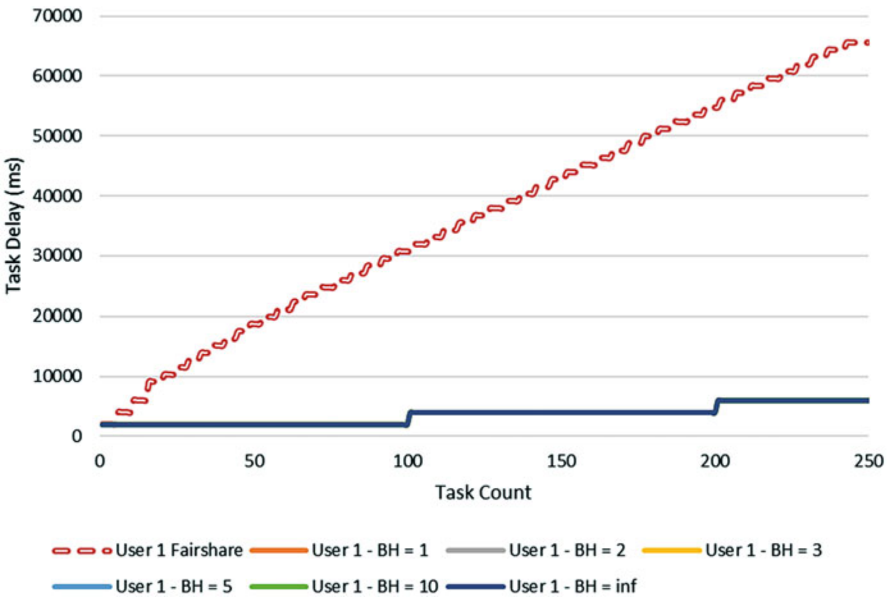


**Fig. 12.6** User 1's task delays with all BH values in simulation 16

## *12.2.3 Simulations 12 and 16 Results and Analysis*

Simulation 12 featured 11 users, and Simulation 16 featured 21 users. In Simulation 12, users 1–10 each submitted 500 tasks, creating a burst of 5000 tasks, while user 11 submitted a one-time load of 50 tasks at PC = 7. A similar pattern was used in simulation 16: each of 20 users submitted 250 tasks in a one-time burst, while user 21 submitted a one-time burst of 50 tasks at PC = 7. This pattern demonstrated the effects of different BH values on task-submission profiles. All 5000 of the tasks submitted by users 1–10 in simulation 12 and users 1–20 in simulation 16 were submitted in bucket 2.

Figure 12.7 shows the delay incurred by one of the smaller users (user 1) in simulation 12, and Fig. 12.6 shows the delay incurred by user 1 in simulation 16. As in the previous simulations and as shown in Fig. 12.8, the First-Come-First-Served (FCFS) feature of the Rawlsian Fair scheduler executed all of the tasks for a given

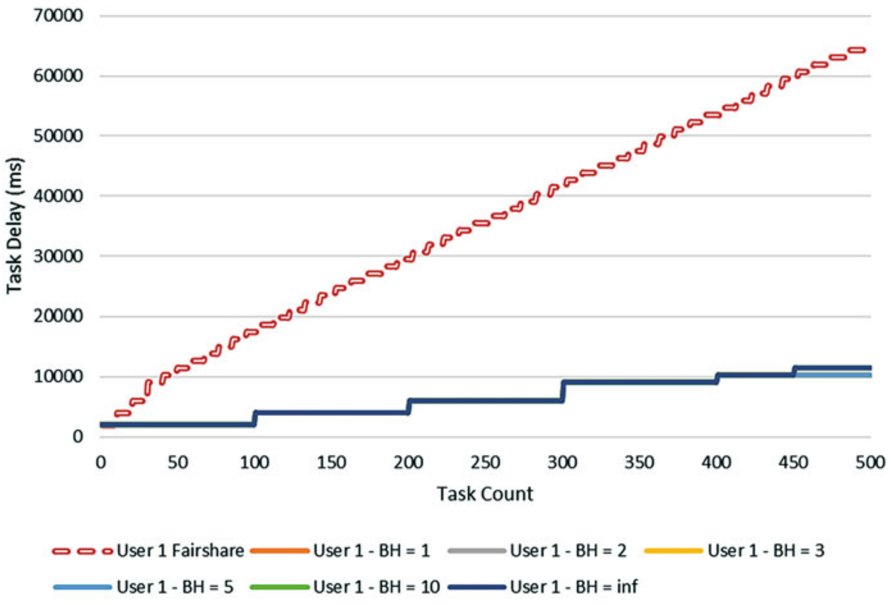


**Fig. 12.7** User 1's task delays with all BH values in simulation 12

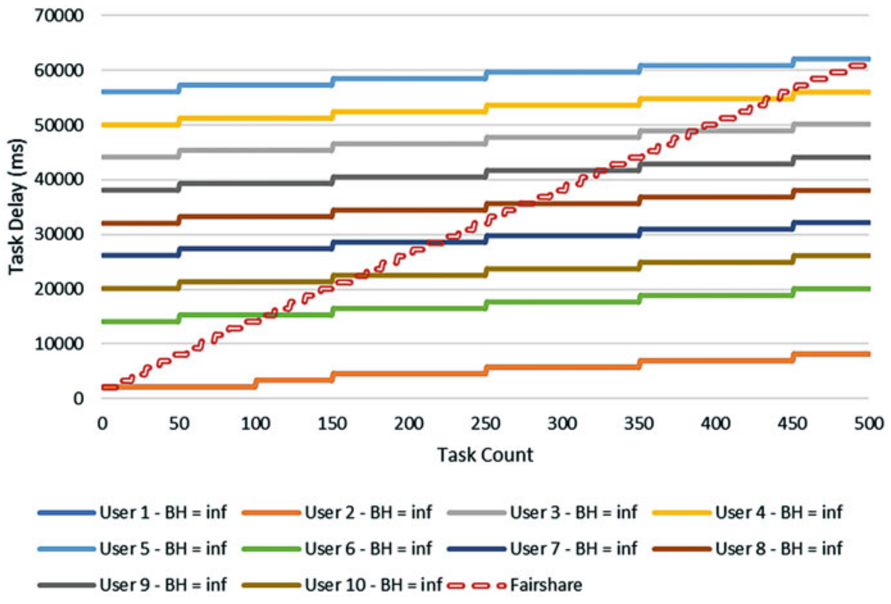


**Fig. 12.8** The task delays experienced by users 1–10 with BH = *inf* in simulation 12

user at once before moving on to the other users. With small BH values of 1, 2, 3, and 5, user 11 and user 21 fell into the FCFS category as well. With BH values of 10 and *inf*, however, these users' tasks were assigned higher seniority values and executed immediately after they were submitted. This effect is shown in Fig. 12.9 and Fig. 12.10. Table 12.5 and Table 12.6 detail the performance gains experienced by user 11 in simulation 12 and by user 21 in simulation 16, respectively. In both cases,

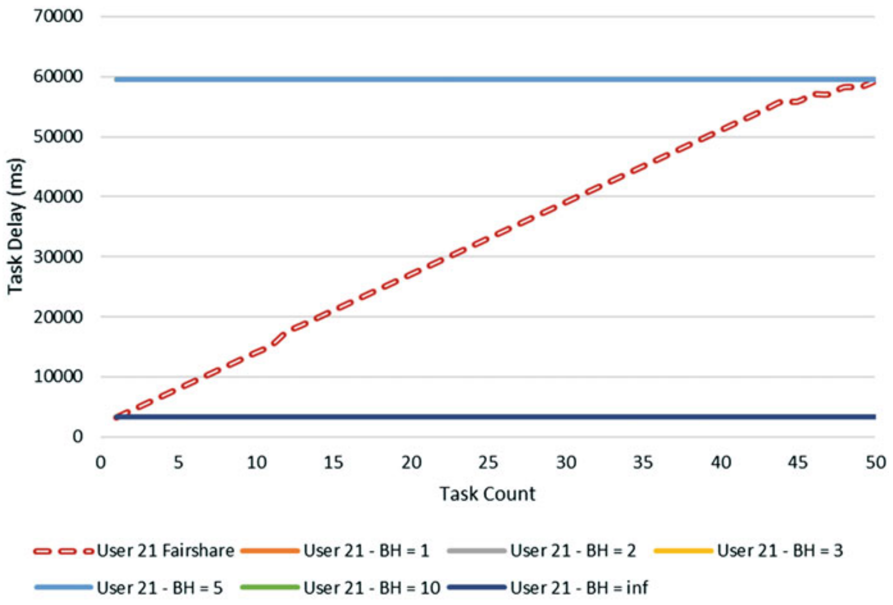


**Fig. 12.9** User 21's task delays with all BH values in simulation 16

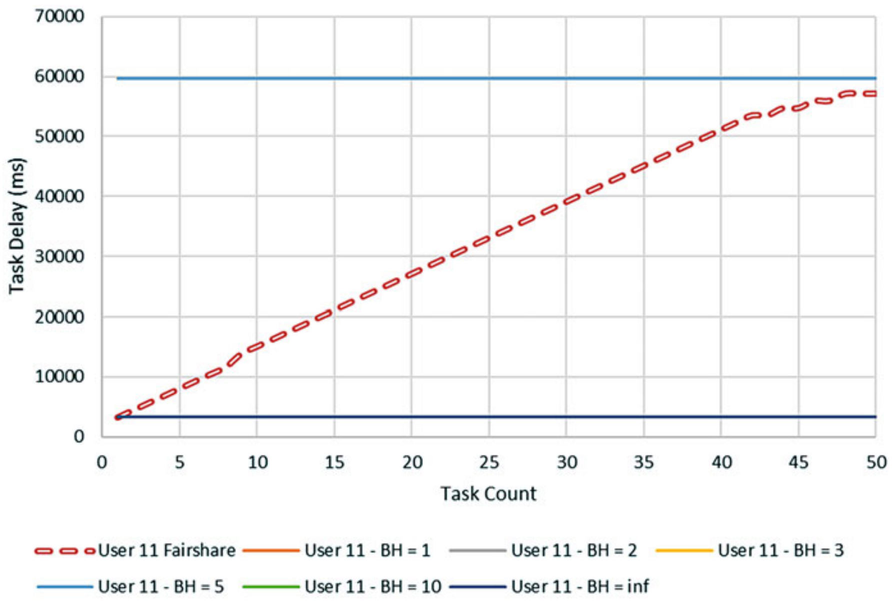


**Fig. 12.10** User 11's task delays with all BH values in simulation 12

**Table 12.5** Comparison of the delays generated by Fair-Share (FS) and Rawlsian Fair (RF) in simulation 12

| | FS delay (ms) | RF delay (ms) BH = 1 | RF delay (ms) BH = 2 | RF delay (ms) BH = 3 | RF delay (ms) BH = 5 | RF delay (ms) BH = 10 | RF delay (ms) BH = inf |
|---|---|---|---|---|---|---|---|
| User 11 | 33,038 | 59,601 | 59,601 | 59,601 | 59,601 | 3301 | 3301 |
| vs fairshare | N/A | −45% | −45% | −45% | −45% | 901% | 901% |

**Table 12.6** Comparison of the delays generated by Fair-Share (FS) and Rawlsian Fair (RF) in simulation 16

| | FS delay (ms) | RF delay (ms) BH = 1 | RF delay (ms) BH = 2 | RF delay (ms) BH = 3 | RF delay (ms) BH = 5 | RF delay (ms) BH = 10 | RF delay (ms) BH = inf |
|---|---|---|---|---|---|---|---|
| User 21 | 33,190 | 59,601 | 59,601 | 59,601 | 59,601 | 3301 | 3301 |
| vs fairshare | N/A | −44% | −44% | −44% | −44% | 905% | 905% |

an improvement of over 900% was seen by the smaller users with BH = *inf*. These improvements are similar to those observed in simulation 4 and simulation 8 (Table 12.1). The dotted red line in the figures represents the delays incurred by the users under fair-share. In these simulations, the tasks submitted by user 11 and user 21 were completed more quickly with higher BH values than under fair-share scheduling. The number of users did not affect the outcomes because these two simulations had results similar to those of Simulation 8.

# Chapter 13
# Class D Results and Simulations

## 13.1 Class D Simulations

Class D workloads represent cases in which one user submits a small, noise-like workload, and one or more disproportionately larger users submit much larger workload. In Simulation 5, one user submits a large workload with 5000 tasks, and another user submits 1 task for every two PCs up to a total of 10 tasks. This submission pattern continues, with 6 users in Simulation 9, 11 users in Simulation 13, and 21 users in Simulation 17. In Simulation 9, users 1–5 submit 1000 tasks each for a total of 5000 tasks. In Simulation 13, users 1–10 submit 500 tasks each, while user 11 submits 5000 tasks on their own. In Simulation 17, users 1–20 submit 250 tasks each.

## 13.2 Class D Results

As Table 13.1 shows, all class D workloads benefit from Rawlsian Fair scheduling. All of the simulations that used Rawlsian Fair performed 17 times (~170%) better than did those that used Fair-Share scheduling.

### *13.2.1 Simulation 5 Analysis and Results*

In Simulation 5, two users had severely disproportionate task-submission profiles. One user submitted a one-time burst of 5000 tasks, while the other user submitted one task every two buckets up to a total of 10 tasks. Figure 13.1 shows the task delay experienced by user 2 with every BH parameter and with fair-share scheduling. User 2 sees as much as 17× (>1600%) increase in performance (Table 13.2).

A. Sedighi, M. Smith, *Fair Scheduling in High Performance Computing Environments*, https://doi.org/10.1007/978-3-030-14568-2_13

**Table 13.1** Class D Submission type average delay

| Class D Average delay (BH = *inf*) | | | | |
|---|---|---|---|---|
| | Simulation 5 | Simulation 9 | Simulation 13 | Simulation 17 |
| User under test | User 2 | User 6 | User 11 | User 21 |
| Fairshare delay (ms) | 53,701 | 54,582 | 54,582 | 54,582 |
| Rawlsian Fair delay (ms) | 3141 | 3141 | 3141 | 3141 |
| Improvement | 1610% ~17× | 1638% ~17× | 1638% ~17× | 1638% ~17× |

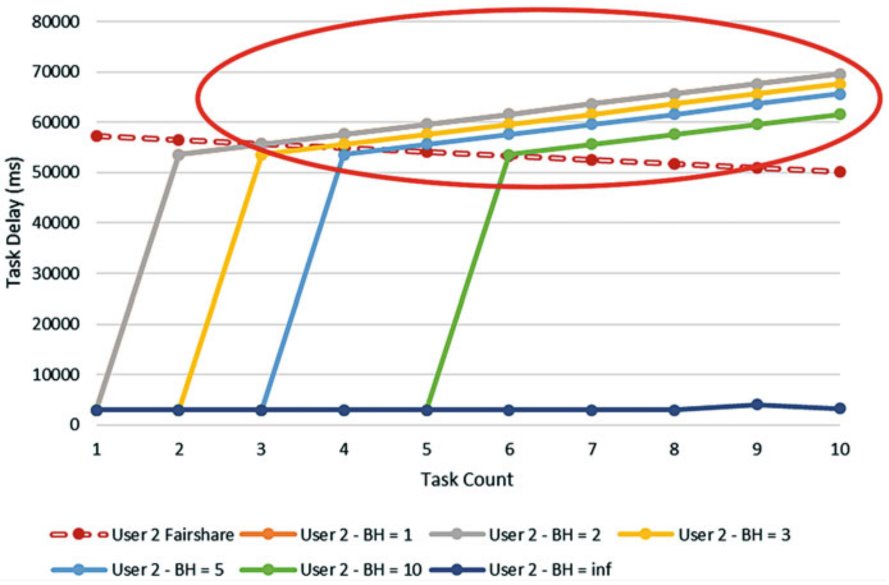


**Fig. 13.1** User 2's task delays with fair-share and all BH values in simulation 5

**Table 13.2** Comparison of the delays generated by fair-share (FS) and Rawlsian Fair (RF) in simulation 5

| | FS delay (ms) | RF delay (ms) BH = 1 | RF delay (ms) BH = 2 | RF delay (ms) BH = 3 | RF delay (ms) BH = 5 | RF delay (ms) BH = 10 | RF delay (ms) BH = inf |
|---|---|---|---|---|---|---|---|
| User 1 | 39,085 | 38,796 | 38,796 | 38,809 | 38,822 | 38,845 | 38,897 |
| vs fairshare | N/A | 1% | 1% | 1% | 1% | 1% | 0% |
| User 2 | 53,701 | 55,741 | 55,741 | 49,081 | 42,621 | 30,301 | 3141 |
| vs fairshare | N/A | −4% | −4% | 9% | 26% | 77% | 1610% |

The red line traversing the top of Fig. 13.1 indicates the task delay experienced with fair-share, which was considerably higher than the task delay experienced with Rawlsian Fair with a bucket history (BH) of infinity (*inf*). The effects of the FCFS

**Table 13.3** Task-Submission timing in simulation 5

| Task number | User 1 submission time (ms) | User 2 submission time (ms) |
|---|---|---|
| 1 | 2000[a] (PC = 2) | 2005 (PC = 2) |
| 2 | 2000 (PC = 2) | 4005 (PC = 4) |
| 3 | 2000 (PC = 2) | 6005 (PC = 6) |
| 4 | 2000 (PC = 2) | 8005 (PC = 8) |
| 5 | 2000 (PC = 2) | 10,005 (PC = 10) |
| 6 | 2000 (PC = 2) | 12,005 (PC = 12) |
| 7 | 2000 (PC = 2) | 14,005 (PC = 14) |
| 8 | 2000 (PC = 2) | 16,005 (PC = 16) |
| 9 | 2000 (PC = 2) | 18,005 (PC = 18) |
| 10 | 2000 (PC = 2) | 20,005 (PC = 20) |
| . . . | | |
| 5000 | 2000 (PC = 2) | N/A |

[a]All simulations start at 2000 ms, or PC = 2. The first 2000 ms, or PC = 1, is used to set up the environment and make sure that everything is ready

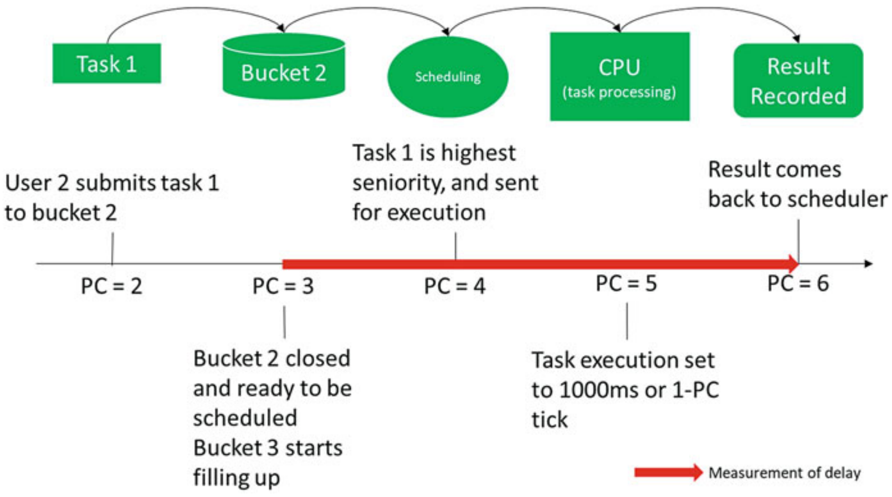

Task 1
Bucket 2
Scheduling
CPU (task processing)
Result Recorded
User 2 submits task 1 to bucket 2
Task 1 is highest seniority, and sent for execution
Result comes back to scheduler
PC = 2
PC = 3
PC = 4
PC = 5
PC = 6
Bucket 2 closed and ready to be scheduled Bucket 3 starts filling up
Task execution set to 1000ms or 1-PC tick
Measurement of delay

**Fig. 13.2** Depiction of how delay in measured in dSim

aspects of the Rawlsian Fair scheduler vis-à-vis bucket history were explained previously and are circled in Fig. 13.1. Simulation 5 produced the lowest possible delay for any single task for the dSim simulator. It is important to discuss in more detail what produced this task-delay value.

User 2's first task was submitted at timestamp 2005, according to Table 13.3, and its results came back at timestamp 6001. The delay was thus approximately 4000 ms, or 4 PC ticks. The cause of this difference was that while within a PC, that bucket is not ready to be scheduled (Fig. 13.2). A given bucket opens at the beginning of a PC and closes at the end of that PC. For this reason, an extra 1000 ms, or 1 PC, was added to the delay.

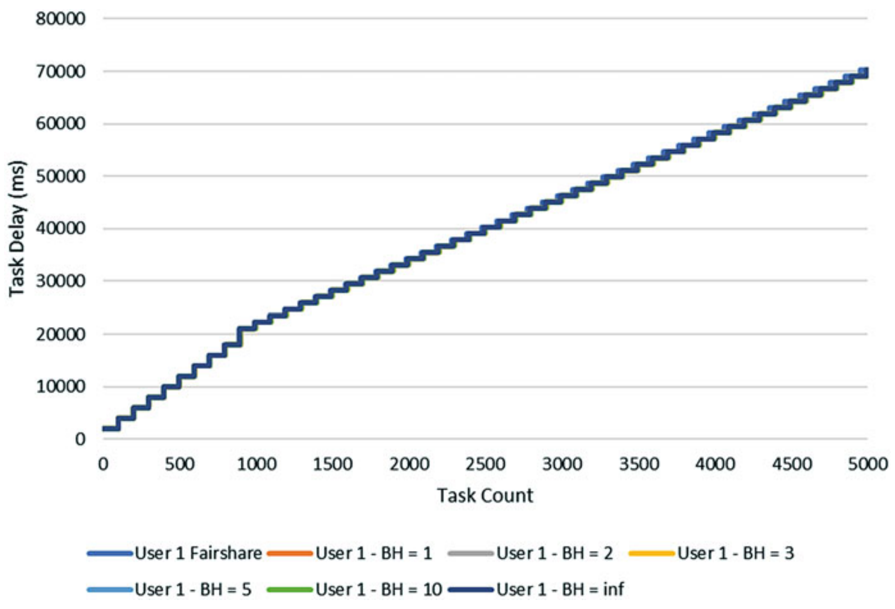


**Fig. 13.3** Task delay for user 1 for Fairshare and all BH values for simulation 5

As is shown in Fig. 13.3, in all scenarios, user 1's performance was unaffected by the scheduling decisions. The task delay that user 1 experienced with each BH value matched the task delay that they experienced under fair-share.

#### 13.2.1.1 Algorithm-Implementation Correctness

The correctness of the implementation is apparent when one considers the results of Simulation 5 with the following parameters for the two users:

$$\mathrm{L}_1(\mathrm{x},\mathrm{PC}) = \max\{5000 * \mathrm{L}_0(\mathrm{x},\mathrm{PC})\cos(\pi * \mathrm{PC}), 0\}, \sum_{PC=1}^{n} L_1(PC) = 5000 \quad (13.1)$$

$$\mathrm{L}_2(\mathrm{x},\mathrm{PC}) = \max\{\ \mathrm{L}_0(\mathrm{x},\mathrm{PC})\cos(\pi * \mathrm{PC}), 0\}, \sum_{PC=1}^{n} L_2(PC) = 10 \quad (13.2)$$

The load for user 1—or $\mathrm{L}_1(\mathrm{x},\mathrm{PC})$ —was 5000 tasks, all of which were submitted at the start of the simulation (eq. 13.1). In contrast, user 2 submitted one task every two PCs up to a total of 10 tasks (eq. 13.2). Table 13.3 further illustrates the task-submission timing in Simulation 5.

User 1 submitted all of their tasks at t = 2000 ms, filling the queue with 5000 tasks in a single PC tick (PC = 2). At t = 2005 ms, after all of user 1's tasks had been submitted, user 2 submitted their first task. After that, user 2 submitted one task every two PCs. In bucket 2, there were 5001 tasks pending execution. In bucket 3, there were no tasks. Between buckets 4 and 20, every other bucket had one task pending execution. The scheduler began accepting tasks at PC = 2.

The dSim simulator is capable of fine-grained timing, and this scenario described a case in which a small user was forced to wait for a larger user; a circumstance commonly known as “the head of the line blocking problem” (Karol et al. 1987). This simulation was conducted with the fair-share scheduling algorithm and the Rawlsian Fair scheduling algorithm with seven different BH sizes.

Different FUD parameters suffered when the BH values were extreme: at BH = 1 and at BH = inf. At BH = 1, dynamicity suffered because the scheduler was unable to process additional tasks until the current bucket had been completed. At BH = inf, fairness suffered because tasks with the same seniority value were executed in a FCFS fashion. In cases in which the users had the same number of tasks pending, only the first user benefited.

#### 13.2.1.2 Expectation for BH = 1 and BH = 2

When the bucket history was set to 1, the scheduler made its decisions using the parameters of the current bucket. The scheduler began taking tasks from the queue at PC = 2 (the 2nd bucket), and user 2’s first task (at PC = 2) was executed quickly. User 1’s 5000 tasks were also in bucket 2, however, since they were submitted at PC = 2. Because the scheduler was programmed to finish each bucket before moving on to the next, we see that subsequently, user 1’s tasks get executed. Once all of user 1’s tasks were completed, the scheduler moved on to the remaining buckets. There was no difference at BH = 2 because user 2 did not submit a task at PC = 3. The next tasks was submitted at PC = 4 Table 13.4.

**Table 13.4** Task delay for Rawlsian Fair with BH = 1 and BH = 2

| Client 2 – task number | Task arrival time (ms) | Task delay time – BH = 1 (ms) | Task delay time – BH = 2 (ms) |
|---|---|---|---|
| 1 | 2005 (PC = 2) | 3001 | 3001 |
| 2 | 4005 (PC = 4) | 53,601 | 53,601 |
| 3 | 6005 (PC = 6) | 55,601 | 55,601 |
| 4 | 8005 (PC = 8) | 57,601 | 57,601 |
| 5 | 10,005 (PC = 10) | 59,601 | 59,601 |
| 6 | 12,005 (PC = 12) | 61,601 | 61,601 |
| 7 | 14,005 (PC = 14) | 63,601 | 63,601 |
| 8 | 16,005 (PC = 16) | 65,601 | 65,601 |
| 9 | 18,005 (PC = 18) | 67,601 | 67,601 |
| 10 | 20,005 (PC = 20) | 69,601 | 69,601 |

**Table 13.5** Task delay for Rawlsian Fair with BH = 3, BH = 5, and BH = 10

| Client 2 – task number | Task arrival time (ms) | Task delay time – BH = 3 (ms) | Task delay time – BH = 5 (ms) | Task delay time – BH = 10 (ms) |
|---|---|---|---|---|
| 1 | 2005 (PC = 2) | 3001 | 3001 | 3001 |
| 2 | 4005 (PC = 4) | 3001 | 3001 | 3001 |
| 3 | 6005 (PC = 6) | 53,601 | 3001 | 3001 |
| 4 | 8005 (PC = 8) | 55,601 | 53,601 | 3001 |
| 5 | 10,005 (PC = 10) | 57,601 | 55,601 | 3001 |
| 6 | 12,005 (PC = 12) | 59,601 | 57,601 | 53,601 |
| 7 | 14,005 (PC = 14) | 61,601 | 59,601 | 55,601 |
| 8 | 16,005 (PC = 16) | 63,601 | 61,601 | 57,601 |
| 9 | 18,005 (PC = 18) | 65,601 | 63,601 | 59,601 |
| 10 | 20,005 (PC = 20) | 67,601 | 65,601 | 61,601 |

#### 13.2.1.3 Expectations for BH = 3, BH = 5, and BH = 10

When the bucket history was set at 3, the scheduler made its decisions using the parameters of the current bucket and of the previous two buckets. For this reason, user 2’s first and second tasks were executed quickly. Because the scheduler was programmed to finish one bucket before moving on to the next, we see that subsequently, user 1’s tasks get executed. Once all of user 1’s tasks were completed, the scheduler moved on to the remaining buckets.

The same pattern was observed with BH = 5; 3 of the tasks were completed quickly. User 1’s tasks followed, after which the scheduler took tasks from the remaining buckets. With BH = 10, 5 of the tasks were completed quickly Table 13.5.

#### 13.2.1.4 Expectation for BH = inf

When the bucket history was turned off (i.e. set to infinity), the scheduler made its decisions using the parameters from the beginning of the simulation. In making its decisions, it considered the entire history of submissions and pending tasks. As was expected, user 2’s tasks were completed quickly, and user 1’s tasks were completed thereafter Table 13.6.

### *13.2.2 Simulation 9 Results and Analysis*

In simulation 9, as in simulation 5, one user had a task-submission profile that was severely disproportionate to those of the other 5 users. Users 1–5 each submitted a one-time burst of 1000 tasks (for a total of 5000 tasks), while user 6 submitted a single task every other bucket up to a total of 10 tasks (Table 13.7).

**Table 13.6** Task delay for Rawlsian Fair with BH = *inf*

| Client 2 – task number | Task arrival time (ms) | Task delay time – BH = *inf* (ms) |
|---|---|---|
| 1 | 2005 (PC = 2) | 3001 |
| 2 | 4005 (PC = 4) | 3001 |
| 3 | 6005 (PC = 6) | 3001 |
| 4 | 8005 (PC = 8) | 3001 |
| 5 | 10,005 (PC = 10) | 3001 |
| 6 | 12,005 (PC = 12) | 3001 |
| 7 | 14,005 (PC = 14) | 3001 |
| 8 | 16,005 (PC = 16) | 3001 |
| 9 | 18,005 (PC = 18) | 3101 |
| 10 | 20,005 (PC = 20) | 3301 |

**Table 13.7** Task submission profiles in simulation 9

| Task count | | | | | | |
|---|---|---|---|---|---|---|
| Submission time (ms) | User 1 | User 2 | User 3 | User 4 | User 5 | User 6 |
| 2000 | 1000 | 1000 | 1000 | 1000 | 1000 | 0 |
| 3000 | 0 | 0 | 0 | 0 | 0 | 1 |
| 4000 | 0 | 0 | 0 | 0 | 0 | 0 |
| 5000 | 0 | 0 | 0 | 0 | 0 | 1 |
| 6000 | 0 | 0 | 0 | 0 | 0 | 0 |
| 7000 | 0 | 0 | 0 | 0 | 0 | 1 |
| 8000 | 0 | 0 | 0 | 0 | 0 | 0 |
| 9000 | 0 | 0 | 0 | 0 | 0 | 1 |
| 10,000 | 0 | 0 | 0 | 0 | 0 | 0 |
| 11,000 | 0 | 0 | 0 | 0 | 0 | 1 |
| 12,000 | 0 | 0 | 0 | 0 | 0 | 0 |
| 13,000 | 0 | 0 | 0 | 0 | 0 | 1 |
| 14,000 | 0 | 0 | 0 | 0 | 0 | 0 |
| 15,000 | 0 | 0 | 0 | 0 | 0 | 1 |
| 16,000 | 0 | 0 | 0 | 0 | 0 | 0 |
| 17,000 | 0 | 0 | 0 | 0 | 0 | 1 |
| 18,000 | 0 | 0 | 0 | 0 | 0 | 0 |
| 19,000 | 0 | 0 | 0 | 0 | 0 | 1 |
| 20,000 | 0 | 0 | 0 | 0 | 0 | 0 |
| 21,000 | 0 | 0 | 0 | 0 | 0 | 1 |

The red line traversing the top of Fig. 13.6 indicates the task delay under fair-share, which was considerably higher than that under Rawlsian Fair with a bucket history (BH) of infinity (*inf*). User 6 experienced performance levels with the Rawlsian Fair scheduler that were up to 17 times better than the performance level they experienced with fair-share (Table 13.8). Varying the bucket size affects how many of the tasks end up getting executed with higher seniority. The same execution pattern was seen in Simulation 5 (13.2.1).

**Table 13.8** Comparison of the delays generated by Fair-Share (FS) and Rawlsian Fair (RF) in simulation 9

| | FS delay (ms) | RF delay (ms) BH = 1 | RF delay (ms) BH = 2 | RF delay (ms) BH = 3 | RF delay (ms) BH = 5 | RF delay (ms) BH = 10 | RF delay (ms) BH = inf |
|---|---|---|---|---|---|---|---|
| User 6 | 54,582 | 55,741 | 55,741 | 49,081 | 42,621 | 30,301 | 3141 |
| vs Fairshare | N/A | −2% | −2% | 11% | 28% | 80% | 1638% |

**Table 13.9** Comparison of the delays experienced by users 1–5 with Fair-Share (FS) and Rawlsian Fair (RF) with BH = *inf* in simulation 9

| | FS delay (ms) | User 1 RF delay (ms) | User 2 RF delay (ms) | User 3 RF delay (ms) | User 4 RF delay (ms) | User 5 RF delay (ms) |
|---|---|---|---|---|---|---|
| BH = inf | 38,939 | 11,203 | 27,821 | 39,821 | 51,821 | 63,821 |
| vs Fairshare | N/A | 248% | 40% | −2% | −25% | −39% |

The effects of the FCFS aspect of the Rawlsian Fair scheduler vis-à-vis bucket history are shown in Fig. 13.5.

While in simulation 5 the task delay for the largest user was unaffected by the smaller user and the delay matched the Fair-Share execution pattern, in simulation 9, each user's task execution was affected by Rawlsian Fair's FCFS methodology. While users 1 and 2 faired better (Fig. 13.5), this came at a cost to users 4 and 5. User 1 saw an increase in performance, while user 5 saw a drop in performance of close to 40% (Table 13.9). For each user, however, changes in the bucket size did not make a difference (Fig. 13.4) because of the large discrepancy between the submission profiles of the larger users (users 1–5) and user 6.

### *13.2.3 Simulations 13 and 17 Results and Analysis*

In simulations 13 and 17, one user's task-submission profile was severely disproportionate to those of many other users. In simulation 13, 10 users each submitted 500 tasks, while one user (user 11) submitted a single task every two PCs up to a total of 10 tasks. A similar submission pattern occurred in simulation 17, but the number of users was increased to 20 and the tasks submitted per user was halved to 250. Because the total number of tasks per type of user (large or small) was the same, it was expected that the Rawlsian Fair scheduler would pick the smaller user out of the crowd in both cases.

Table 13.10 shows the task submission profiles in these two simulations. The left column for each simulation shows the per-user submissions.

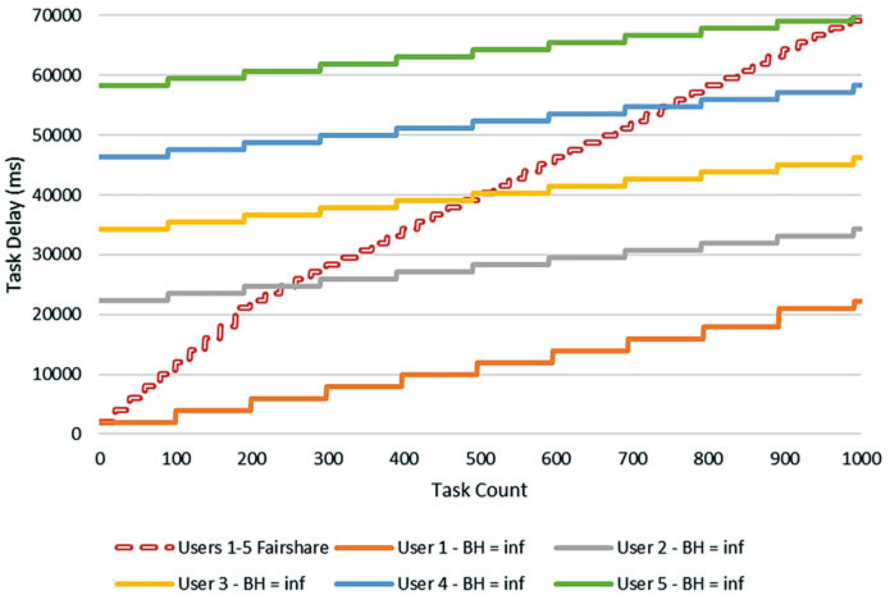


**Fig. 13.4** User 1's task delay with all BH values in simulation 9

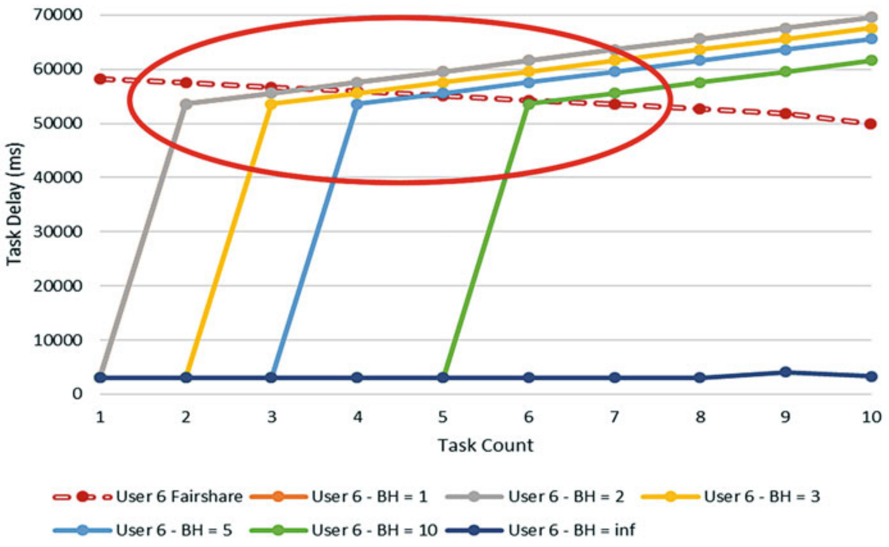


**Fig. 13.5** The task delay experienced by users 1–5 with BH = *inf* in simulation 9

The results of these two simulations are identical, as is shown by Fig. 13.7 and Fig. 13.8. Table 13.11 details the performance improvements experienced by the least-benefitted user (i.e. the smallest user) in both simulations. The identical results obtained in both simulations demonstrate the adaptability of the Rawlsian Fair

**Table 13.10** Task submission profiles for simulations 13 and 17

| | Task count simulation 13 | | Task count simulation 17 | |
|---|---|---|---|---|
| Submission time (ms) | Users 1–10 | User 11 | Users 1–20 | User 21 |
| 2000 | 500 | 0 | 250 | 0 |
| 3000 | 0 | 1 | 0 | 1 |
| 4000 | 0 | 0 | 0 | 0 |
| 5000 | 0 | 1 | 0 | 1 |
| 6000 | 0 | 0 | 0 | 0 |
| 7000 | 0 | 1 | 0 | 1 |
| 8000 | 0 | 0 | 0 | 0 |
| 9000 | 0 | 1 | 0 | 1 |
| 10,000 | 0 | 0 | 0 | 0 |
| 11,000 | 0 | 1 | 0 | 1 |
| 12,000 | 0 | 0 | 0 | 0 |
| 13,000 | 0 | 1 | 0 | 1 |
| 14,000 | 0 | 0 | 0 | 0 |
| 15,000 | 0 | 1 | 0 | 1 |
| 16,000 | 0 | 0 | 0 | 0 |
| 17,000 | 0 | 1 | 0 | 1 |
| 18,000 | 0 | 0 | 0 | 0 |
| 19,000 | 0 | 1 | 0 | 1 |
| 20,000 | 0 | 0 | 0 | 0 |
| 21,000 | 0 | 1 | 0 | 1 |

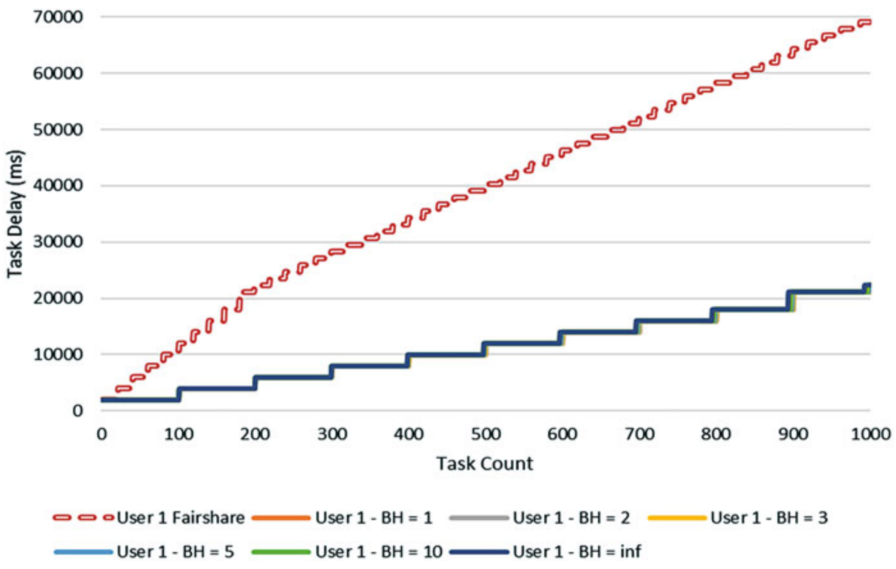


**Fig. 13.6** User 6's task delay with fair-share and all BH values in simulation 9

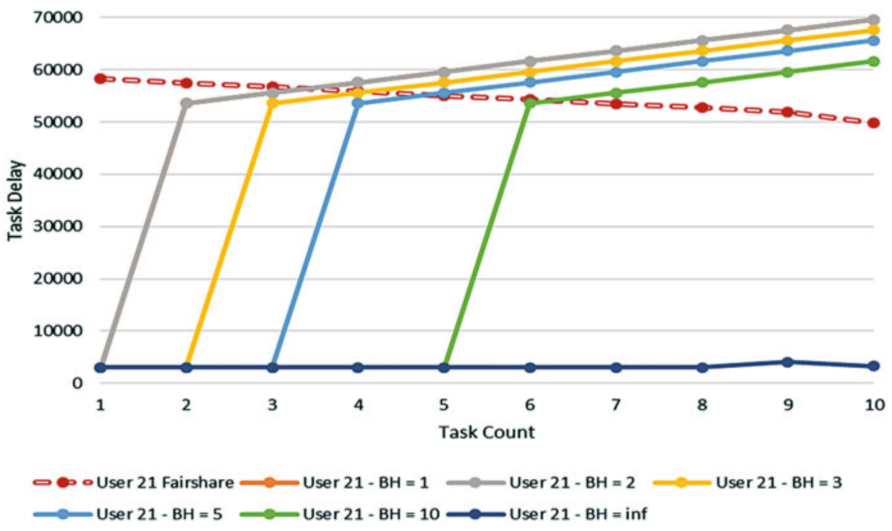


**Fig. 13.7** User 21's task delay with Fair–Share and all BH values in simulation 17

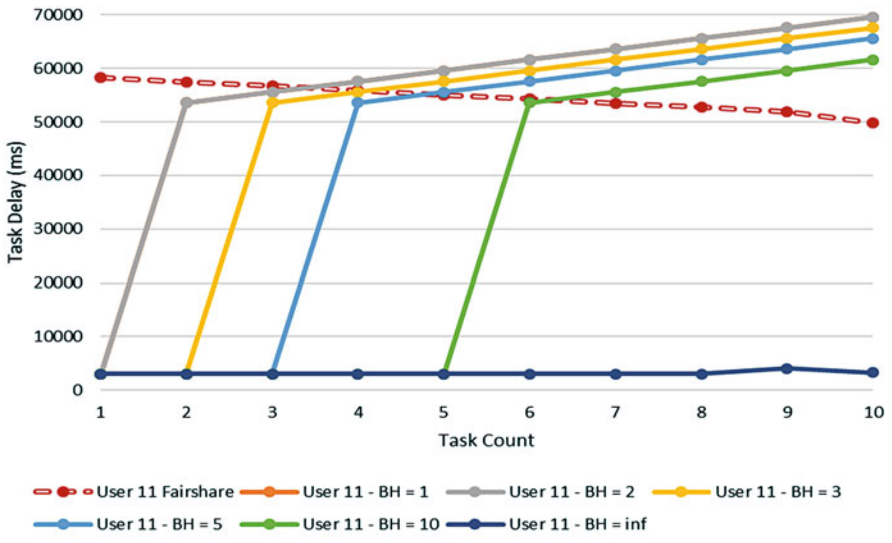


**Fig. 13.8** User 11's task delays with Fair-Share and all BH values in simulation 13

algorithm in detecting the desired user to promote. It can pick the smaller user regardless of the total number of users submitting tasks to the scheduler. The behavior matches the results of simulation 9 (Sect. 13.2.2). The dotted red line traversing the top is the task delay generated by the fair-share scheduling mechanism. The First-Come-First-Served aspect of the Rawlsian Fair scheduling algorithm

**Table 13.11** Comparison of the delays generated by Fair-Share (FS) and Rawlsian Fair (RF) in simulations 13 and 17

| | FS delay (ms) | RF delay (ms) BH = 1 | RF delay (ms) BH = 2 | RF delay (ms) BH = 3 | RF delay (ms) BH = 5 | RF delay (ms) BH = 10 | RF delay (ms) BH = inf |
|---|---|---|---|---|---|---|---|
| User 11 | 54,582 | 55,741 | 55,741 | 49,081 | 42,621 | 30,301 | 3141 |
| vs Fairshare | N/A | −2% | −2% | 11% | 28% | 80% | 1638% |
| User 21 | 54,582 | 55,741 | 55,741 | 49,081 | 42,621 | 30,301 | 3141 |
| vs Fairshare | N/A | −2% | −2% | 11% | 28% | 80% | 1638% |

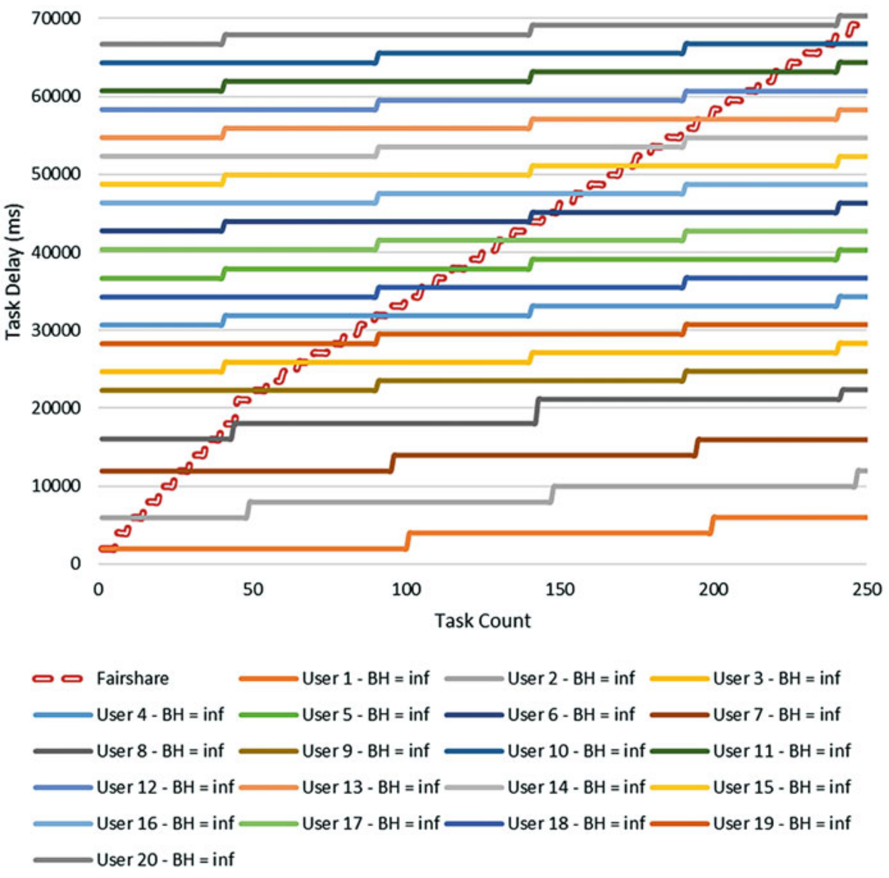


**Fig. 13.9** The task delays experienced by users 1–20 with BH = *inf* in simulation 17

executed the tasks submitted by users 1–10 in simulation 13 and users 1–20 in Simulation 17 in the order in which they were submitted (Fig. 13.9). As was the case with other simulations, some users were better off and some were worse off than they were under fair-share scheduling.

# Chapter 14
# Conclusion

Our work introduced a new scheduler—Rawlsian Fair—which employs Rawls's theory of fairness in scheduling malleable tasks in High Performance Computing environments. This scheduler assigns precedence to the least-well-off user via a new parameter: seniority. Rawlsian Fair performed up to 17× better than did traditional fair-share in scheduling scenarios featuring users with disproportionate task-submission profiles.

The Value-at-Risk problem which is often used by financial services companies to determine the risk associated with a trade represents a typical problem that can benefit from the proposed scheduling methodology. VaR uses the Monte Carlo method, which can be parallelized in HPC environments. Because fair-share scheduling demonstrates shortcomings in handling such problems, a new method that employs the Rawlsian definition of fairness is proposed. This method uses seniority (a new optimization parameter that indicates the time-in-system of each task) to distribute resources in a way that minimizes temporal starvation.

Workloads of four different classes were considered:

– Class A – Complementary Intermittent Workloads. In this class of workload, two users submit intermediate workloads whose distributions complement each other.
– Class B – Steady vs. Intermittent Workloads. In this class of workload, a long-running workload competes with an intermittent workload.
– Class C – Large vs. Small, Transient Workloads. In this type of workload, a large workload competes with a much smaller workload.
– Class D – Large vs. Noise-Like Workloads. In this type of workload, one user's workload is so small relative to another, larger workload that it appears as noise.

These workloads resemble real-world scenarios in which users compete for resources in a shared computing environment. To determine the feasibility of a multi-criteria scheduler vis-à-vis the Rawlsian fairness model, a new simulator—dSim—was built and used to simulate 17 scenarios. These 17 scenarios varied in a number of dimensions, including the number of users (from 2 to 21) and the Bucket

A. Sedighi, M. Smith, *Fair Scheduling in High Performance Computing Environments*, https://doi.org/10.1007/978-3-030-14568-2_14

History. Each simulation was run with 6 different Bucket History (BH) values—1, 2, 3, 5, 10, and infinity—and the results were compared to those achieved with traditional fair-share. Over 600,000 data points were collected. The BH value of infinity (*inf*) proved to be the most beneficial across all simulations. One concern in using a BH value of infinity, however, is that while previous task submissions are never forgotten, submission history may sometimes be irrelevant to current scheduling decisions.

In the Class C and Class D simulations, the Rawlsian Fair scheduler performed between 900% and 1700% (~9× and ~17×) better than traditional fair-share scheduling methods. While users in these types of scenarios have historically suffered under the utilitarian approach to scheduling employed by fair-share schedulers, Rawlsian Fair was shown to reduce the temporary starvation experienced by certain users.

In Class A and class B scenarios, users have similar workloads. In such scenarios with two users, Rawlsian Fair behaves very similarly to fair-share. This indicates that in its worst-case scenarios, Rawlsian Fair performs roughly as well as fair-share. Even though the overall load is the same between simulations, as the number of tasks per user decreases, the Rawlsian Fair scheduler is able to pick users that would have been delayed under fair-share scheduling. On average, class A and B workloads saw an increase in performance of about 80%.

When the FUD (Fairness, Utilization, and Dynamicity) conjecture is considered, increases in fairness come at the cost of either utilization or dynamicity. The decrease in utilization results from the bucketing scheme employed by the Rawlsian Fair scheduler. Because the scheduler completes each bucket before moving on to the next bucket, with lower BH values, utilization may suffer because the number of tasks in a given bucket may not be divisible by the number of available resources. With a BH of infinity, in contrast, dynamicity suffers because the scheduler considers the complete history before making a decision. This may reduce dynamicity when prior history is irrelevant to the decision at hand.

The next step in this line of research is to redesign the scheduler so that BH is dynamic and can be adjusted based on usage and fairness metrics. A random selection method plus using the design methods of BH can provide an alternative to the FCFS limitations of the Rawlsian Fair scheduler, and a data point for comparison to both Rawlsian and traditional fairshare, In addition, dSim can be expanded to enable the graphical representation of real-time information on the performance characteristics of the scheduler.

# References

J. L. Albin, J. A. Lorenzo, J. C. Cabaleiro, T. F. Pena, F. F. Rivera, Simulation of parallel applications in gridSim. *Ibergrid: 1st Iberian Grid Infrastructure Conference Proceedings*, 208–219 (2007)

C. Alexander, Volatility and correlation: Measurement, models and applications. *Risk. Manag. Anal.* **1**, 125–171 (1998)

E. Angel, E. Bampis, F. Pascual, Truthful algorithms for scheduling selfish tasks on parallel machines. *Theor. Comput. Sci.* ***369***(1), 157–168 (2006)

M. Armbrust, A. Fox, R. Griffith, A.D. Joseph, R. Katz, A. Konwinski, et al., A view of cloud computing. *Commun. ACM.* ***53***(4), 50–58 (2010)

M.J. Bach, *The design of the UNIX operating system* (Prentice-Hall, Inc, Upper Saddle River, New Jersey, 1986a)

M.J. Bach, *The design of the UNIX operating system*, vol 5 (Prentice-Hall, Englewood Cliffs, 1986b)

K.R. Baker, D. Trietsch, *Principles of sequencing and scheduling* (Wiley, New Jersey, 2013)

W.H. Bell, D.G. Cameron, A.P. Millar, L. Capozza, K. Stockinger, F. Zini, Optorsim: A grid simulator for studying dynamic data replication strategies. *Int. J. High. Perform. Comput. Appl.* ***17***(4), 403–416 (2003)

J. Bentham, *An introduction to the principles of morals and legislation* (Clarendon Press, Oxford, 1879)

L.v. Bertalanffy, *General system theory; foundations, development, applications* (G. Braziller, New York, 1969)

D. Bertsimas, V.F. Farias, N. Trichakis, The price of fairness. *Oper. Res.* ***59***(1), 17–31 (2011)

P. Brucker, B. Jurisch, B. Sievers, A branch and bound algorithm for the job-shop scheduling problem. *Discret. Appl. Math.* ***49***(1), 107–127 (1994)

P. Brucker, S. Knust, *Complex scheduling*, 2nd edn. (Springer, Heidelberg, 2012)

H. V. Bui, *Fairshare scheduling-a case study*, University of Arkansas, (2008)

R. Buyya, M. Murshed, GridSim: A toolkit for the modeling and simulation of distributed resource management and scheduling for Grid computing. *Concurrency. Comput. Pract. Experience.* ***14*** (13–15), 1175–1220 (2002). https://doi.org/10.1002/Cpe.710

J. Cao, S. A. Jarvis, S. Saini, G. R. Nudd, *Gridflow: Workflow management for grid computing*. Paper presented at the Cluster Computing and the Grid, 2003. Proceedings. CCGrid 2003. 3rd IEEE/ACM International Symposium on

A. Chervenak, I. Foster, C. Kesselman, C. Salisbury, S. Tuecke, The data grid: Towards an architecture for the distributed management and analysis of large scientific datasets. *J. Netw. Comput. Appl.* ***23***(3), 187–200 (2000)

J. Chong, K. Keutzer, M.F. Dixon, Acceleration of market value-at-risk estimation. *Available at SSRN**1576402*** (2009)

G. Christodoulou, L. Gourves, F. Pascual. Scheduling selfish tasks: about the performance of truthful algorithms *Computing and Combinatorics,* (Springer, 2007), pp. 187–197

B. Committee, *Basel III: A global regulatory framework for more resilient banks and banking systems* (*Basel Committee on Banking Supervision*, *Basel*, 2010)

R.W. Conway, W.L. Maxwell, L.W. Miller, *Theory of scheduling* (Courier Corporation, NewYork, 2012)

A. Demirguc-Kunt, E. Detragiache, O. Merrouche, Bank capital: Lessons from the financial crisis. *J. Money. Credit. Bank.**45***(6), 1147–1164 (2013)

F. Dong, S. G. Akl, Scheduling algorithms for grid computing: State of the art and open problems: Technical report (2006)

F.Y. Edgeworth, The hedonical calculus. *Mind**4***(15), 394–408 (1879)

M. N. Baily, D. J. Elliott, The US financial and economic crisis: Where Does It Stand and Where Do We Go From Here?: Business and Public Policy at BROOKINGS (2009).

D.H. Epema, J. de Jongh, Proportional-share scheduling in single-server and multiple-server computing systems. *ACM SIGMETRICS Perform. Eval. Rev.**27***(3), 7–10 (1999)

D. Feitelson, L. Rudolph, U. Schwiegelshohn, K. Sevcik, P. Wong, Theory and practice in parallel job scheduling, in *Job scheduling strategies for parallel processing*, ed. by D. Feitelson, L. Rudolph, vol. 1291, (Springer, Heidelberg, 1997), pp. 1–34

D. Ferraioli, C. Ventre, *On the price of anarchy of restricted job scheduling games.* Paper presented at the ICTCS (2009)

I. Foster, C. Kesselman, *The Grid 2: Blueprint for a new computing infrastructure*: (Elsevier 2003)

N. Georgescu-Roegen, Utility. *Int. Enc. Soc. Sci.**16***(1), 236–267 (1968)

P. Glasserman, P. Heidelberger, P. Shahabuddin, *Efficient monte carlo methods for value-at-risk,* (2010)

S. Gleeson, *International regulation of banking: Basel II, capital and risk requirements* (Oxford University Press Catalogue, Oxford, 2010)

H. Hakenes, I. Schnabel, Bank size and risk-taking under Basel II. *J. Bank. Financ.**35***(6), 1436–1449 (2011)

M. Harchol-Balter, *Performance modeling and design of computer systems: Queueing theory in action* (Cambridge University Press, Cambridge, 2013)

G.J. Henry, The unix system: The fair share scheduler. *AT&T Bell. Labs. Tech. J.**63***(8), 1845–1857 (1984)

H. Hoogeveen, Multicriteria scheduling. *Eur. J. Oper. Res.**167***(3), 592–623 (2005)

H. Hussain, S.U.R. Malik, A. Hameed, S.U. Khan, G. Bickler, N. Min-Allah, et al., A survey on resource allocation in high performance distributed computing systems. *Parallel. Comput.**39*** (11), 709–736 (2013)

H. A. James, *Scheduling in metacomputing systems* Citeseer (1999)

J. Janyška, M. Modugno, R. Vitolo, Semi--vector spaces and units of measurement. *arXiv preprint arXiv:0710.1313,* (2007)

P. Jorion, *Value at risk* (McGraw-Hill, New York, 1997)

M. Karol, M. Hluchyj, S. Morgan, Input versus output queueing on a space-division packet switch. *IEEE Trans. Commun.**35***(12), 1347–1356 (1987)

J. Kay, P. Lauder, A fair share scheduler. *Commun. ACM.**31***(1), 44–55 (1988)

S. D. Kleban, S. H. Clearwater, *Fair share on high performance computing systems: What does fair really mean?* Paper presented at the Cluster Computing and the Grid, 2003. Proceedings. CCGrid 2003. 3rd IEEE/ACM International Symposium on (2003)

D. Klusacek, Alea 3.0 web site, 11/17/2014., from http://www.fi.muni.cz/~xklusac/alea/

D. Klusáček, H. Rudová, *Alea 2: Job scheduling simulator.* Paper presented at the Proceedings of the 3rd International ICST Conference on Simulation Tools and Techniques (2010)

R. B. Krishnamurthy, I. Chin, A. Chinnapatlolla, *Exploration of parallelization frameworks for computational finance.* Paper presented at the Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA) (2012)

R. Layard, Income and happiness: Rethinking economic policy, Lecture (2003)

R. D. Luce, H. Raiffa, *Games and decisions: Introduction and critical survey*. (Courier Corporation 2012)

S. Marston, Z. Li, S. Bandyopadhyay, J. Zhang, A. Ghalsasi, Cloud computing—The business perspective. *Decis. Support. Syst.**51***(1), 176–189 (2011)

R. McGill, *Technology management in financial services*: (Springer, 2008)

A.J. McNeil, R. Frey, P. Embrechts, *Quantitative risk management: Concepts, techniques and tools* (Princeton university press, Princeton, New Jersey, 2015)

J. F. Nash Jr, The bargaining problem. *Econometrica: Journal of the Econometric Society*, 155-162 (1950)

J. Norstad, An introduction to utility theory (1999). *Unpublished manuscript at*http://homepage.mac.com/j.norstad

V. Pareto, *Manuale di economia politica con una introduzione alla scienza sociale (manual of political economy)* (*Societa Editrice Libraria*, *Milano*, 1919)

R. G. Parker, *Deterministic scheduling theory.* (CRC Press, 1996)

C. Potts, L.N. van Wassenhove, Single machine tardiness sequencing heuristics. *IIE Trans.**23***(4), 346–354 (1991)

J. Rawls, *Justice as fairness: A restatement* (Harvard University Press, Cambridge, 2001)

J. Rawls, *A theory of justice.* (Harvard university press, 2009)

D. Read, Utility theory from jeremy bentham to daniel kahneman (2004)

C. Reyes, K. Walters, W. Yang, *Monte carlo within a day.* Paper presented at the quantitative analysis in financial markets: Collected papers of the New York university mathematical finance seminar (2001)

Rockwell Automation, (2016), 10/26/2016, from https://www.arenasimulation.com/

G. Sabin, G. Kochhar, P. Sadayappan, (2004 15–18 Aug. 2004). *Job fairness in non-preemptive job scheduling.* Paper presented at the Parallel processing, 2004. ICPP 2004. International Conference on

A. Sedighi, Y. Deng, P. Zhang, Fariness of task scheduling in high performance computing environments. *Scalable Computing: Pract. Experience**15***(3), 273–285 (2014). https://doi.org/10.12694/scpe.v15i3.1020

A. Sedighi, M. Smith, Y, Deng, (2017a, November 3rd–5th 2017). *An evaluation of optimizing for FUD in scheduling for shared computing environments.* Paper presented at the 2nd IEEE International Conference on Smart Cloud (SmartCloud 2017), New York

A. Sedighi, M. Smith, Y. Deng, *FUD – Balancing scheduling parameters in shared computing environments*. Paper presented at the 4th IEEE International Conference on Cyber Security and Cloud Computing (IEEE CSCloud 2017) (New York 2017b)

R.A. Shiner, Aristotle's theory of equity. *Loy. LAL Rev.**27***, 1245 (1993)

S. Soltesz, H. Pötzl, M. E. Fiuczynski, A. Bavier, L. Peterson, *Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors.* Paper presented at the ACM SIGOPS operating systems review (2007)

R. Sugden, Rational choice: A survey of contributions from economics and philosophy. *Econ. J.**101*** (407), 751–785 (1991)

V. T'kindt, J.-C. Billaut, *Multicriteria scheduling: theory, models and algorithms* (Springer Science & Business Media, Berlin, Heidelberg, 2006)

A. Takefusa, S. Matsuoka, O. Tatebe, Y. Morita, *Performance analysis of scheduling and replication algorithms on grid datafarm architecture for high-energy physics applications.* Paper presented at the High Performance Distributed Computing, 2003. Proceedings 12th IEEE International Symposium on (2003)

D.K. Tarullo, *Banking on Basel: The future of international financial regulation* (Peterson Institute, Washington, D.C., 2008)

F. W. Taylor *The principles of scientific management* (1911)

S. Tezuka, H. Murata, S. Tanaka, S. Yumae, Monte Carlo grid for financial risk management. *Futur. Gener. Comput. Syst.* ***21***(5), 811–821 (2005)

J. Von Neumann, O. Morgenstern, *Theory of games and economic behavior* (Princeton university press, Princeton, 2007)

C. A. Waldspurger, W. E. Weihl, Lottery scheduling – flexible proportional-share resource management. *Operating Systems Design and Implementation (Osdi)*, 1–11 (1994)

C. A. Waldspurger, W. E. Weihl, *Stride scheduling: Deterministic proportional share resource management*: Massachusetts institute of technology. Laboratory for computer science (1995)

M.R. Weisbord, *Productive workplaces revisited: Dignity, meaning, and community in the 21st century* (Wiley, Hoboken, 2004)

L.H. White, *How did we get into this financial mess?* (*CATO Institue Briefing Papers*, Washington, D.C., 2008)

L. H. White, Housing finance and the 2008 financial crisis. *CATO Institute* (2009)

A. Wierman, Fairness and scheduling in single server queues. *Surv. Oper. Res. Manag. Sci.* ***16***(1), 39–48 (2011)

H.P. Young, *Equity: in theory and practice* (Princeton University Press, Princeton, 1995)

# Index