Supercomputer Emulation For Evaluating Scheduling Algorithms

Claudio Barberato

A thesis submitted for the degree of
Master of Philosophy
The Australian National University

June 2018

Except where otherwise indicated, this thesis is my own original work.

Claudio Barberato
10 June 2018

to my family, wife Fabiana Soares Santana and son Andre Santana Barberato.

Acknowledgments

I would like to thank my supervisory panel, Dr Eric McCreath, Associate Professor Peter Strazdins and Dr Muhammad Atif for their indispensable guidance, without which the completion of this thesis would not be possible.

Abstract

Scheduling algorithms have a significant impact on the optimal utilization of HPC facilities, yet the vast majority of the research in this area is done using simulations. In working with simulations, a great deal of factors that affect a real scheduler, such as its scheduling processing time, communication latencies and the scheduler intrinsic implementation complexity are not considered. As a result, despite theoretical improvements reported in several articles, practically no new algorithms proposed have been implemented in real schedulers, with HPC facilities still using the basic first-come-first-served (FCFS) with Backfill policy scheduling algorithm.

A better approach could be, therefore, the use of real schedulers in an emulation environment to evaluate new algorithms.

This thesis investigates two related challenges in emulations: computational cost and faithfulness of the results to real scheduling environments.

It finds that the sampling, shrinking and shuffling of a trace must be done carefully to keep the classical metrics invariant or linear variant in relation to size and times of the original workload. This is accomplished by the careful control of the submission period and the consideration of drifts in the submission period and trace duration. This methodology can help researchers to better evaluate their scheduling algorithms and help HPC administrators to optimize the parameters of production schedulers. In order to assess the proposed methodology, we evaluated both the FCFS with Backfill and Suspend/Resume scheduling algorithms. The results strongly suggest that Suspend/Resume leads to a better utilization of a supercomputer when high priorities are given to big jobs.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Thesis Statement

The emulation of a supercomputer environment is a more powerful alternative for evaluating scheduling algorithms than simulation because the first automatically takes into account the scheduling processing time, communication latencies, multi programming and fragmentation. Emulation is viable if the processing time of a workload can be reduced by means of sampling and time shrinking.

1.2 Problem Statement

A supercomputer is a high-level performance computer composed of thousands of processing units or cores organized into nodes that are connected by a state-of-art network. Supercomputers are specially designed to carry out the computation of large problems called jobs. These big problems require multiple processing units for their timely completion.

A scheduler is a program that decides when and where a job is going to run. This problem grows in complexity when the number of cores requested by incoming jobs exceeds the capacity of the supercomputer [Feitelson and Rudolph, 1995; Feitelson et al., 1997; Feitelson, 1997; Feitelson et al., 2005].

A good scheduler will optimize the use of the resources, e.g., processing units, resulting in a better return for the High Performance Computing (HPC) investment. This makes research into scheduling algorithms a hot topic in computer science. The problem is complex because it requires dealing with various, sometimes conflicting, requirements such as the number of cores, total of requested and used memory, priorities among many others [Lifka, 1995].

The usual approach in developing scheduling algorithms involves the use of simulators [Schwiegelshohn and Yahyapour, 1998]. Traditionally, simulators are relatively simple programs developed by researchers to test their hypotheses. A simulator usually reads a file with the workload of a supercomputer facility to sort the jobs according to established and newly proposed algorithms. The basic information contained in these workloads is the arrival time, the number of requested nodes and the estimated and run time of each job. The outcome of these simulations may contain

information about when a job was submitted, started, finished and the used nodes. This information can be used to evaluate scheduling metrics like average waiting time, average response time and overall system throughput. These metric values form, then, the basis for performance comparisons between scheduling algorithms.

However, despite theoretical improvements reported in several articles [Yuan et al., 2014; Niu et al., 2012; Niemi and Hameri, 2012], practically no new proposed algorithms have been implemented in real schedulers with most HPC facilities still using the basic first-come-first-served (FCFS) with backfill policy scheduling algorithm [Schwiegelshohn and Yahyapour, 1998].

In the present work, instead of disregarding all aspects related to the resource management, the chosen approach was to use a real system in an emulation environment to overcome the limitations of the simulators. The emulation environment was composed of a cluster of virtual machines managed by a real scheduler and running dummy jobs which were responsive to suspend and continue signals.

1.3 Scope

There are several aspects affecting the evaluation of job scheduling in supercomputers, such as the scheduling processing time, workload structure, communication latencies, communication within a job, memory availability, interactive jobs, multi programming level inside a processing unit, fragmentation, warm up and cool-down periods. A great deal of these factors are usually not considered when working with simulations.

In the present work, due to the fact that emulations are closer to reality, most of these aspects are automatically considered, like scheduling processing time, communication latencies, multi programming and fragmentation. Additionally, the warm up and cool-down periods are also taken into account in our experiments.

However, this thesis does not investigate the workload structure, communications within a job, memory availability and interactive jobs.

1.4 Contributions

This thesis uncovers the related challenges to perform timely emulations which results can be safely extrapolate to the original workload. The main contributions of this thesis are:

- The introduction of a simple, but effective, framework to support the evaluation of scheduling algorithms.
- The development of an efficient program, *sigsleep*, that plays the role of an incoming job. This program uses a small amount of memory, demands little CPU time and it is responsive to the suspension and continue signals sent by the scheduler. It logs all data related to its run independently of the scheduler,

which allows the direct evaluation of the performance metrics. The source code of this program is presented in Appendix C.

- The development of a submission script that reads a workload and precisely submits *sigsleep* jobs to the scheduler.
- The development of a program to evaluate the classical metrics based on the information logged by *sigsleep*.
- The study of four artefacts, concurrency, submission order, sampling and time shrinking that affect the scheduling process and how to overcome their effects on the evaluation of the scheduling metrics.
- The use of the developed framework to evaluate the implementation of two popular scheduling algorithms, Backfill and Suspend/Resume, in Slurm. We found that Suspend/Resume shows a better performance in comparison to Backfill only when big jobs are presented in the workload.
- The uncovering of a lack of optimization in the Suspend/Resume implementation in Slurm when used in conjunction with the Consumable Resources plugin.

1.5 Thesis Outline

This thesis has 6 chapters. Chapter 1 states the problems this thesis will address, defines its scope and outlines its contributions.

Chapter 2 introduces the most relevant concepts related to supercomputers, clusters, cloud computing and classical scheduling metrics. It also presents a literature review of the current research on scheduling algorithms using both simulation and emulation.

Chapter 3 explains how the environment used in our experiments was set up. It also describes the hardware and software platforms used to create this environment, along with the algorithms and programs developed to evaluate scheduling algorithms.

Chapter 4 discusses the experimental methodology applied in the experiments performed in this work. This chapter also introduces the key concepts of the classical metrics artefacts.

The results of the application of the aforementioned methodology for the evaluation of two scheduling algorithms by means of the emulation are presented in Chapter 5.

Finally, Chapter 6 summarises the contributions of this thesis for the evaluation of scheduling algorithms, and identifies possible future developments in this research area.

Chapter 2

Background and Related Work

This chapter provides the background knowledge required to develop this thesis. It also contains a comprehensive literature review about simulation, emulation, scheduling algorithms development and the strategies for their evaluation. This chapter starts by introducing the basic concepts related to this work in Sections 2.1 and 2.2, and then progresses to explaining supercomputer scheduling in Section 2.3. Section 2.4 gives a brief introduction to classical scheduling metrics while Section 2.5 defines and compares simulation and emulation. Sections 2.6 and 2.7 review and discuss previous works on the usage of simulation and emulation to evaluate scheduling algorithms.

2.1 Supercomputer, Computer Cluster and Cloud Computing

A supercomputer is a high-level performance computer composed of thousands of processing units designed for HPC. A computer cluster is a set of processing units, called nodes, connected by a network and used as a single computer. The use of high-speed networks and powerful microprocessors associated with computer clusters can turn them into very cost effective supercomputers [Baker and Buyya, 1999].

Cloud computing is a way of sharing computational resources over the Internet. It is implemented by adding a layer of hardware virtualization called hypervisor that, among many other functionalities, creates, destroys and runs virtual machines (VM). Cloud computing is, therefore, a flexible and cost-effective computational environment where computational resources can be allocated on demand. Nevertheless, this flexibility may come at the expense of a deteriorated performance due to the extra work demanded by the hypervisor.

However, due to recent developments in hardware and software, cloud computing is being increasingly considered as an alternative platform for HPC [Atif et al., 2016]. In a cloud computing environment, instead of having different jobs sharing a single real cluster, we have single jobs running in dedicated VM clusters. In this scenario, the costs of performing HPC work would be smaller compared to a dedicated HPC facility.

2.2 Resource Management Systems

Resource Management Systems (RMS) administrate the utilization of supercomputer resources, such as CPU, memory, storage, and network. One of the most important tasks of RMS is scheduling jobs submitted by users. For this reason, RMS are often simply called schedulers in the literature, which is the terminology adopted for this thesis. As examples of RMS we can cite Load Sharing Facility (LSF) [IBM, 2017], and IBM Load Lever [Yonghong and Chapman, 2008], both systems developed by IBM, Portable Batch Systems (PBS) produced by Altair [Alt, 2017], and Simple Linux Utility for Resource Management (Slurm) [Jette et al., 2003].

2.3 Scheduling

Supercomputer scheduling is a very popular research topic as shown by the large number of publications in the area. Almost all of the published works resort to simulations to test their hypotheses and, as a probable consequence of this approach, few of the proposed algorithms were actually implemented in real schedulers [Schwiegelshohn and Yahyapour, 1998].

Such a relevant research topic demands a periodic review of the state of the field from time to time. In 1995, [Feitelson and Rudolph, 1995] and again in 1997 [Feitelson et al., 1997; Feitelson, 1997] defined precisely the basic concepts of scheduling and the requirements that should be satisfied by the scheduler. They proposed the use of different configurations along the day because the user's expectations about a submitted job change accordingly with the time of submission. The intractability of many scheduling problems was also discussed, including preemptive and non-preemptive gang scheduling. [Feitelson, 1997] cited a number of studies that have demonstrated that despite the overheads of preemption, the flexibility derived from the ability to preempt jobs allows for much better schedules. The most often quoted reason for using preemption is that time slicing gives priority to short running jobs and, therefore, approximates the Shortest-Job First (SJF) policy. In 2005, [Feitelson et al., 2005] reviewed again the status of parallel job scheduling. This paper covered the two main algorithmic approaches, Backfilling and Gang Scheduling, used by real supercomputer installations. The paper also discussed successful and unsuccessful approaches to this problem. Its main thesis is that, despite the actual usage patterns largely remained within the realm of batch scheduling, the progress on this field has led to significant improvement in the utilization of supercomputer resources, going from 50-70% in the past to 90% of utilisation by 2005.

The most basic scheduling algorithm is the traditional First-Come-First-Serve (FCFS). In this case, the incoming jobs are allowed to use the supercomputer nodes only in the strict order of their submission. When there are not enough idle nodes left to fulfill the requirements of an incoming job, the scheduler put it in a queue, as well as all subsequent jobs. The number of idle nodes increases as the running jobs finish their run. Once the number of idle nodes is enough, the scheduler allows the execution of the first job in the queue.

EASY (Extensible Argonne Scheduling sYstem) Backfilling scheduling algorithm [Lifka, 1995], is an optimisation of the First Coming First Serve (FCFS) scheduler. EASY requires a user's estimate for the job runtime. Accordingly to this algorithm, smaller jobs further in the queue can be scheduled earlier if they do not delay the starting time, also known as reservation time, of the job on the head of the queue.

One variant of EASY is Conservative Backfilling, where a later job can jump the queue only if it does not delay any other earlier job in the queue. There is also a compromise between these two, where only a certain number of reservations is guaranteed. It was found that guaranteeing the reservation for the first four jobs is a good compromise. EASY is highly dependent on the estimates of the running time of each job and the overestimation of this parameter actually helps the performance of backfilling. This is because the net effect of overestimation is to make backfilling behaves more like SJF schedulers.

Preemption is a scheduling technique where running jobs can be preempted (suspended) and queued jobs can use the now freed nodes. Usually, the preempted jobs can later resume its activities on the same node.

Suspend/Resume algorithms use preemption to obtain a more optimised utilisation of the resources. In its most simple form, high priority queued jobs preempt low priority running jobs. As shown in this thesis, even this relatively simple policy can lead to a better utilization if the priority is related to the job's size. In this case, big jobs will preempt small jobs as soon as they become the first job in the queue. This simple technique avoids the need for the accumulation of idle nodes in order to run a big job.

Gang scheduling, another algorithm mentioned in [Feitelson et al., 1997; Feitelson, 1997], preempts jobs in all nodes. Other jobs previously preempted in the same nodes, are then allowed to run for a determined slice of time. This algorithm displays the particular feature of preventing small jobs from being held in the queue by long ones. The drawback of this approach is the time wasted by the system to preempt all jobs across all nodes. Flexible algorithms have been proposed where I/O activity is monitored and only jobs with complementary characteristics are paired in the same processors. Despite its theoretical advantages over backfilling, gang scheduling was only successfully implemented on the Connection Machine CM-5. However, several efforts have been made to overcome the inherited drawbacks of gang scheduling by providing hardware that supports faster context switching.

[Schwiegelshohn, 2014] presents a set of guidelines to design a scheduling algorithm. The main motivation of this work was the fact that most research publications reviewed by the authors had a negligible impact in real schedulers. Its main thesis was that the definition of a few general rules for presenting new algorithms could help in their implementation in real schedulers. This work shows that since the introduction of EASY backfilling in 1995, despite the large number of research papers, the authors were not aware of any actual implementations of the suggested algorithms. The article identifies two probable reasons for the lack of interest by the practitioners: 1) the proposed algorithms were not practicable, or 2) improper communication between authors and system developers. Another important reason, though not di-

Table 2.1: Scheduling parameters.

Parameter	Description
l	Wall time
t	CPU time
a	Submission time
s	Starting time
c	Completion time
e	Estimate run time
r_t	Response time
r	Run time
u	Suspension time
p	Priority
b	Number of processors (cores)

rectly mentioned by this paper, is that scheduling in practice is very complex, and the building and modifications of an efficient and stable scheduler requires a huge amount of time, money and talent.

2.4 Classical scheduling metrics

For the purposes of this thesis, we shall consider the workload as the data about each job i submitted to a supercomputer during a certain time period. Basically, the workload contains the submission time, a_i, the number of requested cores, b_i, the priority, p_i and the estimated runtime, e_i and actual runtime, r_i, of each job during the period. This workload, processed by the scheduler, is called then a supercomputer trace. This trace contains, beyond the information already provided in the workload, the starting, s_i, and completion time, c_i. Wall time, l_i, is the elapsed time as determined by a wall clock. CPU time, t_i, is the time consumed by a job and it might differentiate from wall clock due, for instance, to time-sharing and suspension. If suspend/resume technique is used, the trace also stores u_i, which is the total time a job is suspended. The classical scheduling metrics are average values calculated based on the times found in a trace and they are frequently used to compare the performance of different scheduling algorithms. When comparing algorithms, we also need to introduce the concept of fairness. Considering the jobs in a queue, an algorithm is considered fair if no job with lower priority is allowed to run before a job with higher priority. The algorithm can still be considered fair if the running of the low priority job does not affect the starting time of the job with higher priority, as in the case of Backfill. Table 2.1 summarizes all the main parameters used in this thesis.

Based on the aforementioned values, the most common performance scheduling

metrics can be easily evaluated. The total time, which is time elapsed from the submission of the first job until the completion of the last job of the trace, takes the form

$$T = c_{last\_job} - a_{first\_job} \tag{2.1}$$

and it is the most basic metric that can be used to compare two scheduling algorithms. Algorithm A will be considered superior to algorithm B, provide fairness is kept, if $T_A < T_B$. That is the case, for instance, of FCFS with backfill when compared with the basic FCFS.

The Deadline d_i is the maximum wall time that a job is allowed to run, and it is given by

$$d_i = s_i + u_i + e_i \tag{2.2}$$

For a trace with n jobs, the Average Waiting Time,

$$w_t = \sum_{i=1}^{n} \frac{s_i - a_i}{n} \tag{2.3}$$

basically measures how long a job will stay in the queue. It has been proved [Conway et al., 1967] that, given a workload, the strategy of scheduling the shortest job first (SJF) will result in the lowest values for this metric.

The Response Time,

$$r_t = \sum_{i=1}^{n} \frac{c_i - a_i}{n} \tag{2.4}$$

takes also into account the runtime of each job and measures the time that takes, on average, for users to obtain their results.

Long jobs have a disproportional influence in the response time. The slowdown,

$$s_d = \sum_{i=1}^{n} \frac{(c_i - a_i)/r_i}{n} \tag{2.5}$$

tries to minimize the influence of these long jobs by dividing the response time of each job by its corresponding runtime.

Weighted Slowdown,

$$w_s = \sum_{i=1}^{n} \frac{(c_i - a_i)b_i/r_i}{n} \tag{2.6}$$

is a metric that prioritises jobs requiring a big number of cores by multiplying the slowdown of each job by the number of cores it requires. This is an important metric because the core objective of a HPC facility is solving complex problems that requires a considerable number of cores per job.

In a cluster with C cores, the cluster utilization,

$$u_t = \sum_{i=1}^{n} \frac{r_i \, b_i}{C \, T} \tag{2.7}$$

is a metric that evaluates the percentage of the cluster that was effectively used during the period T.

2.5 Simulation and Emulation

Simulators and emulators try to model, in different levels, the behaviour and the internal workings of a real system.

In a simulator, the functionalities of the real system are highly abstracted. For example, scheduling decisions, unlike on a real system, can occur instantaneously. The internal working of these functionalities in the simulator usually does not reflect how things are implemented on a real system.

An emulator is an intermediate approach between a real system and a simulator. An emulator, like the simulator, can also offer only a few functionalities of the real system, but it tries to reproduce as close as possible the internal workings of the system.

As an example of a real system, we can think about an old valve radio. One can write a software that runs on a computer that simulates the appearance of the radio on the screen. If the power button of the radio simulator is turned clockwise, the radio would turn on and the volume would increase. The real implementation of this feature could bear no resemblance with how things work in a real old radio.

In an emulator, the internal working of some parts of a radio would be mimicked as closely as possible. In this case, we can imagine that the radio emulator wouldn't turn on the radio immediately because a real old radio would need some time to warm up its valves.

According to [McGregor, 2002], an emulator is a model where some working functions of the model can also be accomplished by the use of components of a real system.

A real HPC system is composed of a supercomputer with one manager node, several processing nodes and a scheduler. Users log on to the manager node and submit their jobs through the scheduler. The scheduler, based on the current state of the running and queued jobs, decides then when and where a job will run.

Simulators built to test scheduling algorithms can be much simpler because most of the details of a real scheduler can be ignored. In the simplest case, the simulator reads a file with a supercomputer trace and applies to its jobs some scheduling algorithms. Classical scheduling metrics are then used to compare the performance of the proposed and well-established algorithms. Contrastingly, emulators environments are much more complicated and difficult to build because they try to use, as much as possible, real components.

Some schedulers, like Moab, PBS-pro and Slurm, provide a simulation mode, where the running of a trace can be performed without the real execution of jobs.

This can be classified as emulation since the scheduler operates similarly to the real system.

In this thesis, an emulator was built using 11 virtual machines of an Openstack cloud environment connected together via NFS and running a real scheduler. The emulations were performed by means of a script that reads a trace and runs for each job a special sleep program.

2.6 Scheduling Research Using Simulation

Being the easiest way of testing an idea, most of the research in new algorithms is done by means of simulations. The lack of real implementations of algorithms, as discussed below, is a clear evidence of the considerable distance between the development of an idea by means of simulation and its actual implementation in a real scheduler.

This section presents a literature review of the most relevant solutions proposed to improve scheduling research using simulation.

The user runtime estimates have a great impact on how the scheduling occurs. The EASY approach is more efficient when users overestimate, by a great degree, the runtime estimates [Mu'alem and Feitelson, 2001], [Feitelson et al., 2005]. This happens because the overestimation provides a very long reservation time for the first job in the queue. As the running jobs finish well before their expected time, large holes in the scheduling are formed, which can be filled by shorter jobs. This has the net effect of approximating EASY to FCFS [Tsafrir and Feitelson, 2006].

Checkpointing is a technique where a job periodically saves its state in a file. If this job is stopped by any reason before its normal ending, it can later resume its work from the last saved checkpoint. A Checkpoint-based FCFS backfilling algorithm [Niu et al., 2012] presents improvements of up to 40% in the most common metrics. In this algorithm, the backfill is done more aggressively, i.e., filling the holes left by FCFS with jobs that, according to its runtime estimates, would violate the reservation time for the first job in the queue. Due to the user's notorious tendency of overestimating the runtime of their jobs, most of these jobs finish before the reservation time comes. The few ones that would be still running after this time would be checkpointed, killed and put back in the queue.

Given enough memory in the nodes, Suspend/Resume technique can also be used to explore the idle nodes left by FCFS [Snell et al., 2002]. In this case, instead of killing a job that is violating a reservation time, this job is simply suspended. A more sophisticate algorithm, called PV-EASY [Tsafrir and Feitelson, 2006], proposes two new strategies named Shadow Load Preemption (SLP) and Venture Backfilling (VB). Usually, it is not possible to guarantee fairness without compromising performance and vice-versa. When done aggressively, backfilling can delay queued jobs for a very long period. SLP backfilling scheduling, on the other hand, guarantees strict fairness while VB is used to improve its performance. In the proposed algorithm, when either a new job arrived or a job finishes, PV-EASY firstly uses the EASY

algorithm. The set of the highest priority running jobs are called sunny load. The set of jobs with priorities lower than any waiting jobs is called shadow load. A backfilled job can change from the shadow to the sunny state during its lifetime. The shadow load is the main cause of unfairness. One approach to solve this problem is to allow the preemption of shadow jobs if this preemption enables higher priorities jobs in the queue to start immediately. A venture backfilling was devised in order to minimize the change of preemption and, consequently, improve system utilisation. This strategy involves the computation of the reservation time of the first job in the queue, a system-generated runtime prediction and careful selection of a job with the biggest probability of successful termination. According to the authors, PV-EASY guaranties fairness, good performance and does not violate reservation times.

Most of the proposed scheduling algorithms focus on improving the so called classical performance metrics, Equations 2.3-2.6. Notably, although being always considered an important feature, fairness has rarely been included as an optimisation criterion for a HPC scheduler. The fair start time (FST) metric [Klusácek and Rudová, 2012] evaluates the influence of arriving jobs on the reservation time for the jobs already in the queue. The unfairness is evaluated as the difference between the FST, which is the reservation time for each job disregarding all later jobs and the actual start time. Another contribution is an optimisation procedure designed to improve the performance of the Conservative Backfilling algorithm. This heuristic uses three parameters, the schedule that will be optimised, the maximum number of iterations and a time limit. In each iteration, one job is randomly removed from its current position and the schedule is immediately compressed while an overall metric, which is a combination of the classical and FST metrics, is calculated. This job is inserted for a period in a *taboo* list, which means that it is prevented for a while to be selected again. The experimental evaluation presented by the authors demonstrated that the proposed extension represented a significant improvement by means of fairness and performance over several existing algorithms, including FCFS, Conservative and EASY backfilling, as well as aggressive backfilling without reservations.

A responsiveness metric, the Schedule Length Ratio (SLR), takes into account the networking delays [Burkimsher et al., 2013]. Through simulations, it was shown that a proposed algorithm that minimizes this metric delivers better responsiveness and fairness than other schedulers that do not rely on user runtime estimates.

Another metric, the thinking time, which is the interval between the response and submission of two consecutive jobs submitted by a single user, can play an important role in the scheduler decisions [Schlagkamp, 2015]. Several simulations showed that this happens because changes in the scheduling will change, in turn, the user's think time.

A somewhat middle term solution between a simple simulator and an emulator was proposed by [Lucero, 2011] in the form of a simulation mode for the Slurm scheduler. The main thesis behind this work was that a real resource manager/scheduler has several variables that need to be tuned to optimize the utilization of a HPC facility. It also emphasizes that the scheduling is a two-step process where first a job is selected and then the available resources are matched with it. Most simulators

completely ignore the second step leading to results that are not valid in real situations. Therefore, a simulation mode where both steps are present, could help Slurm administrators to find the optimum value for scheduling parameters. This simulation mode could also help Slurm developers in the development of scheduling algorithms. The solution presented in this paper tried to make minimum code changes in Slurm so developers and users would have all Slurm features in the simulation mode. The modifications used the LD PRELOAD functionality of shared objects in UNIX systems. This feature made possible to capture specific function calls exported by shared objects and to replace code when necessary. LD PRELOAD was used to modify the calls to time() function returning a simulated time instead of real time. By using this approach, a trace could run 31 times faster than the original runtime.

[Lucero, 2011] also mentioned that the Moab cluster scheduler supported a simulation mode although it has never worked as described. The reason behind such behaviour was the complexity and high costs associated with keeping the simulation mode in the newer versions. This also has proven to be the case of the simulation mode in Slurm since the development of this feature was abandoned soon after its inception and it is no longer maintained.

2.7 Scheduling Research Using Emulation

[Tsafrir and Feitelson, 2006] used a cluster of 32 compute nodes plus one management node to study the influence of two strategies used in schedulers, backfilling and gang scheduling, and the effects of the multiprogramming level. Their results show that the best result was achieved by a combination of both backfilling and gang scheduling with a limited multiprogramming level due to finite memory. One interesting feature of this study was the need of reducing all times from the used workload by a constant factor in order to run the experiments faster. The chosen shrinking factor, 100, was given without any justification and with no analysis of its implications. Contrastingly, this thesis performs a comprehensive study about the implications of shrinking time on the stability of the scheduling metrics, aiming to develop a methodology where this kind of propositions are fully justifiable.

An improved throughput and energy consumption can be achieved through the increasing of memory utilisation, i.e., running more than one job per computing node [Niemi and Hameri, 2012]. This was done at the Worldwide LHC Computing Grid at the Large Hadron Collider at CERN. Due to the fact that this is a HPC facility exclusively dedicated to analyse LHC data, the submitted jobs could be categorised into three groups: CPU-intensive, memory intensive and I/O-intensive. It was proved that running both I/O and CPU intensive jobs in the same node improves both throughput (10-20%) and energy consumption (5-20%). The emulations were performed using a front-end computer, 1 GB network and 2 nodes with 6 CPU cores, 16 gigabytes of memory and 1 TB hard disk each.

[Soner and Ozturan, 2013] implemented an auction-based algorithm, called AUCSHED, that was used in a heterogeneous CPU-GPU scheduler plug-in. The algorithm

generates multiple bids for a window of jobs at the beginning of the queue. The algorithm maximizes an objective function that uses the jobs priorities. The same authors, in [Soner and Ozturan, 2015], developed a topological aware job scheduling for Slurm. Slurm documentation discourages the configuration of the exact network topology because it leads to slower scheduling. The proposed scheduling method overcomes this limitation and improve the job intercommunication by a judicious job allocation based on the real network topology. In both papers, the authors utilized an emulation mode for Slurm.

More recently, [Rodrigo et al., 2017] presented a scheduling simulation framework that encloses all the steps of the scheduling development, providing workload modeling and generation, system simulation, comparative analysis and experiment orchestration. The framework utilizes 17 computing nodes and achieves up to 15 speed-ups over real-time. The authors stated that the limiting factor of the simulations speed-up is the scheduling runtime which depends on the number of jobs in the waiting queue. Speed-ups of the same order were also obtained in this thesis without the use of a simulator but our conclusion about the limiting factor differs. Since the number of jobs in the queue was kept constant during our experiments, the limiting factor found was the job submission rate. When this rate exceeds Slurm capacity to process jobs, the cluster nodes will be idle for longer which compromises the evaluation of the classical scheduling metrics. These findings will be discussed in more detail in Section 4.2.4.

2.8 Summary

Despite the popularity of the research in supercomputer scheduling, only a few of the proposed algorithms were actually implemented in real schedulers. The main reason behind this fact is the distance between the relative simplicity of simulations and the complexity of a real supercomputer environment. In the next chapter we will present the design and implementation of what we believe is a better environment and methodology to study and evaluate scheduling algorithms.

Chapter 3

Design and Implementation

This chapter provides a general overview of the design of the cluster used to emulate a supercomputer as well as the programs and algorithms developed to obtain the results presented in this thesis. Implementations details are given in the software and hardware sections.

3.1 Computing Cluster

The design and implementation of the cluster used in this thesis are presented in the next subsections. Subsection 3.1.1 describes the minimum steps to set up a cloud-based cluster. Subsection 3.1.2 shows how to establish a Slurm based cluster at the National Computational Infrastructure (NCI) Nectar Cloud. Finally, using this infrastructure, Subsection 3.1.3 presents a way to emulate a supercomputer size cluster.

3.1.1 Cloud-Based Cluster

A cluster can be set up in a cloud environment in a variety of ways depending on where it is hosted. The most common first step is to create a VM which is going to serve as the cluster head node. This step is followed by the installation and configuration of software packages for parallel computing and communication, like the OpenMP, MPI and *ssh* based log in. Computing nodes can then be created using this head node as a template. Adding the IP addresses of each the computing nodes to the appropriate configuration files in the head node creates a cluster capable of running parallel jobs.

Additionally, a network file system can be mounted in all VMs so the updating, installation and running of packages and programs for resource management can be easily achieved.

3.1.2 NCI Slurm Based Cluster

The National Collaboration Tools and Resources project (Nectar) is an initiative of the Australian Government to provide computational power to Australian researchers

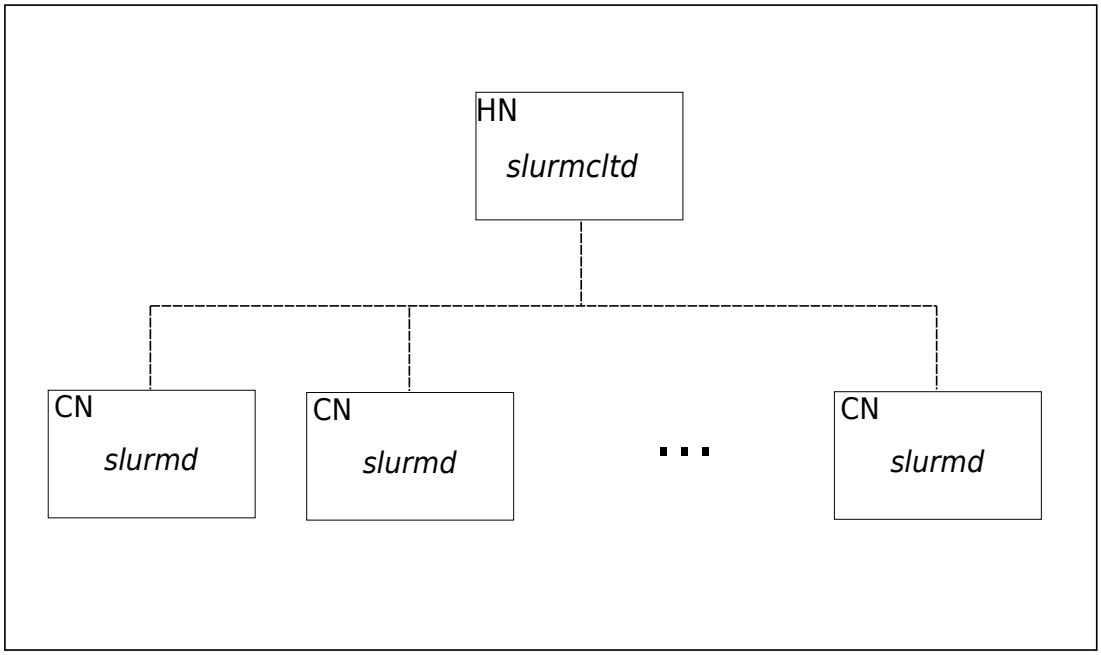

Figure 3.1: A normal Slurm based cluster configuration: One *slurmctld* daemon running in the Head Node (HN) and one *slurmd* daemon running in each Computing Node (CN).

through the establishment of a cloud computing infrastructure. Nectar allows researches to easily build software and services on the cloud. The Nectar infrastructure is provided through a partnership among research facilities in Australia where NCI is one of its eight nodes.

Nectar uses OpenStack as its cloud software platform. OpenStack is free and open-source software and consists of several components written in Python to control pools of processing, storage, and networking resources. Users can manage it through a web-based dashboard, command-line tools, or web services.

Slurm (Simple Linux Utility for Resource Management), [Jette et al., 2003], is an open-source distributed resource management system software for Linux clusters. Slurm manages the access to computing resources, starts, executes and monitors jobs, and schedules jobs. The two basic components of Slurm architecture are *slurmd*, which is a daemon running on each compute node (CN) and responsible for starting each job, and *slurmctld*, a daemon running on a head node (HN).

Munge [Dunlap, 2004] is a software package for the creation, authentication, and validation of credentials in an HPC cluster. Slurm uses Munge to secure the communication between its main components. Each time a *slurmd* wants to communicate with the *slurmctld*, it first requests a credential from Munge in the local node. This credential is packed together with the message and send to the head node. Likewise, a communication received by *slurmctld* is only processed if the credential contained in the receiving message is validated by Munge in the head node.

[Atif, 2016] developed a series of scripts and Python programs to automate the steps of building a cloud-based cluster on the Nectar cloud. We used this to build the

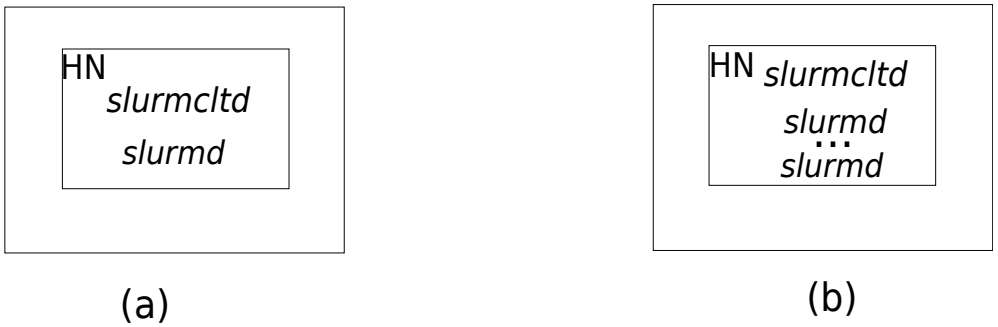

Figure 3.2: Cluster emulation using Slurm Front End and Multiple slurmds features. (a) A single *slurmctld* and single *slurmd* can run in the Head Node (HN), or in a desktop, if Slurm is compiled with the Front End option. In this case, the emulated nodes will be set up in the Slurm configuration file but all job management will be accomplished by the single *slurmd*. (b) Additionally to the Front End option we can also run multiples *slurmds* in a single computing node with the Multiple Slurmd option. As in (a) the emulated nodes will be also set up in the configuration file, but the management of all the jobs will be shared between the several *slurmds*.

cluster showed in figure 3.1 with one head node and ten processing nodes running CentOS 6.8. All VMs in this cluster have a m2.medium flavour which is how Nectar characterizes a VM. A m2.medium flavour is a VM with 2 CPUs, 6 GB of RAM and 30GB of disk. A volume with 10 GB of space was attached to these nodes. This volume is a storage entity, which is independent of the cluster and can be re-attached to any other VM. The group X is a standard security group in the Nectar cloud and it determines the rules set in the iptables of each VM. The cluster configuration is summarized in Table 3.1.

3.1.3 Cluster Emulation

The emulation of a cluster bigger than its actual size can have many applications. For instance, such bigger clusters can make a better use of its processing power if the jobs running on it are not CPU intensive [Niemi and Hameri, 2012]. The emulation can also help the development of scheduling algorithms. In this case, a sufficiently large cluster could run a supercomputer workload which might provide more realistic results to the developer.

Slurm allows a single node to appear as a cluster by enabling the front end option (Figure 3.2(a)). This permits the running of one *slurmd* in the same node as the *slurmctld*. Emulated nodes can then be set up in the Slurm configuration file but care must be taken to avoid overloading the slurmd daemon with too many simultaneous jobs. The front end option allows the running of Slurm in a single node or in a desktop computer, but this feature was not used in this thesis because we wanted to isolate the working of the *slurmctld* from any other sources that could compete for CPU resources.

Another way of emulating a bigger cluster in Slurm is allowing the running of multiple *slurmds* on a single node. This can be achieved during the compilation time

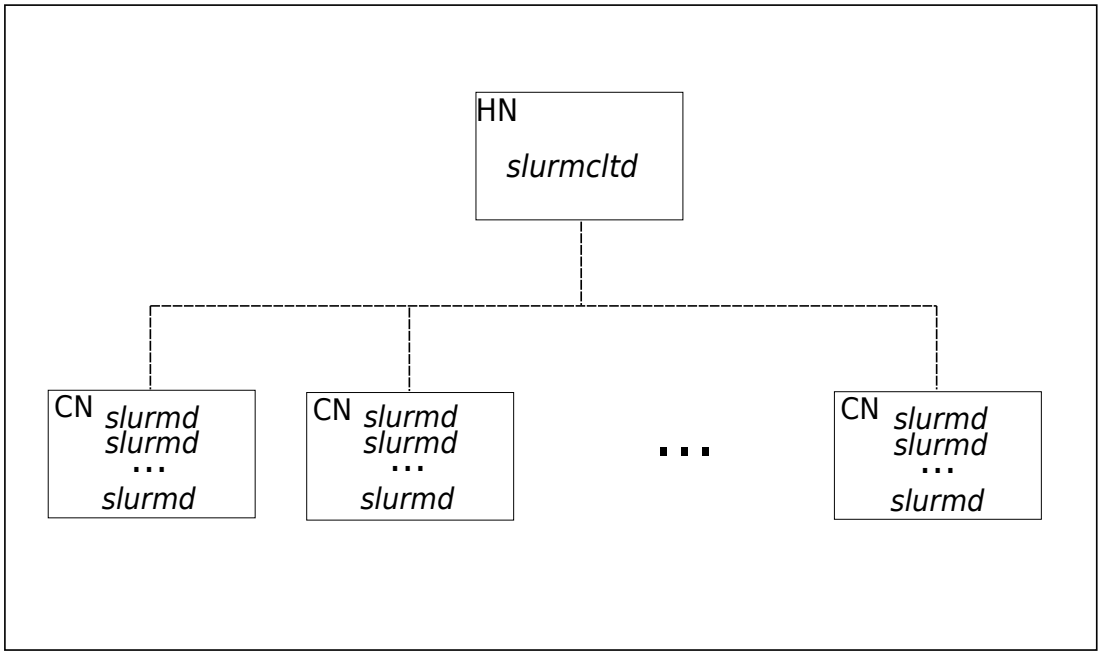

Figure 3.3: A cluster emulation with a *slurmctld* in the head node (HN) and multiples *slurmds* in each computing node (CN).

by the use of the *–enable-multiple-slurmd* option. In combination with the Front End option, we can achieve a configuration like the one showed in Figure 3.2(b), i.e., one *slurmctld* and several *slurmds* running in a single node.

A greater number of emulated CPUs per node can be achieved by simply configuring more CPUs per node in the Slurm configuration file (*slurm.conf*). Additionally, the parameter *FastSchedule* needs to be set to 2 so Slurm will only consider the configuration of each node that is specified in the configuration file.

Using only the multiple *slurmds* per node feature, as shown in Figure 3.3, and setting more CPUs per node than the actual number, we obtained a cluster with 360 *slurmds* per node with each one of them configured with 16 CPUs. This combination was sufficient for the emulation of a 3,600 node/57,600 CPU computing cluster. This configuration was very close to NCI's supercomputer, Raijin, which, by the time the experiments shown in this thesis were performed, had 3,592 nodes and 57,472 cores.

Throughout this thesis, in accordance with Slurm terminology, we will treat CPUs and cores as synonyms.

3.2 Hardware platform

The NCI's Nectar cloud provides access to a very fast cloud environment composed of Intel Xeon Sandy Bridge 2.6 GHz processors with 16 cores in 2 sockets, 64GB of memory per node, solid state disk storage, 56GbE network and redundant storage [Atif et al., 2016].

Table 3.1: Cluster parameters.

Parameter	Value
Name	clusterANU
Operating System	CentOS 6.8
Flavor	m2.medium
Virtual CPUs per Node	2
Memory per Node	6 GB
Number of Nodes	10
Availability Zone	NCI
Security group	X
Additonal Securities	port 111
Slurmds Per Node	360
Configured CPUs per Node	16
Total Slurmds	3600
Total apparent CPUs	57600

3.3 Main Programs

The main programs and scripts developed during this thesis are presented in the next subsections. *Sigsleep* is a program designed to play the role of a job. It responds to eventual suspensions and writes all context data in a log file. *Runtrace* is a script developed to automate the submission of a workload. For each entry in the workload file, it submits a *sigsleep* program to the cluster. *Calcmets* is a program which uses the data stored in the log file created by *sigleep* processes to calculate the classical scheduling metrics.

3.3.1 SigSleep

The emulation of a supercomputer involves the simultaneous running of thousands of programs. A small part of these programs are related to the management of the cluster, e.g. the scheduler, while the vast majority represents incoming jobs. In order to fit all these programs in a finite amount of processors and memory, a dummy program, preferentially one that consumes very little memory and CPU, may be used to play the role of a job. The most common choice is the *sleep* program which is readily available in the *bash* environment. However, this program is not affected by suspensions as illustrated in Appendix A. As we are using scheduling algorithms that rely on suspensions another solution was required.

This motivated the development of the program *sigsleep*. *Sigsleep* (Signal Sleep) is a program that plays the role of a job during the emulation of a supercomputer's workload. It has a small memory footprint and demands little CPU time. Yet, unlike

Algorithm 3.1: A sleep program responsive to suspensions.

```
1  struct {
2      double suspend, resume;
3      suspend_period *next;
4  } suspend_period;
5  struct {
6      int id, n_cores, n_suspensions;
7      timespec submission, start, end, runtime;
8      suspend_period *sus_res;
9  } job_data;
10 suspend_period *sus_per;
11 Function sighandler(signum)
       Input  : One nonnegative integer signum
       Output : The instant the program was suspend
12     sus_per ← new(suspend_period);
13     clock_gettime(suspend);
14     sus_per.suspend ← suspend;
15     signal(signum, SIG_DFL);
16     kill(getpid(), signum);
17 Program sigsleep(id,priority,n_cores,submission,runtime)
       Input  : 2 integers(id, n_cores), 2 timespec (submission, runtime) and 1
                string (priority)
       Output : A entry in the log file with the submission, start, end and
                suspensions times
18     job_data job;
19     clock_gettime(job.start);
20     job ← id, priority, n_cores, submission;
21     job.n_suspensions ← −1;
22     job.sus_res ← empty_list;
23     remain ← runtime;
24     do
25         signal(SIG_TSTP, sighandler);
26         job.n_suspensions++;
27         if job.n_suspension > 0 then
28             clock_gettime(resume);
29             sus_per.resume ← resume;
30             add sus_per to the list job.sus_res;
31         runtime ← remain;
32     while nanosleep(runtime, remain) == −1;
33     clock_gettime(job.end);
34     write(LOG_FILE, job);
```

sleep, *sigsleep* keeps track of the suspension periods by detecting and dynamically storing the instants the suspension and resume signals that are sent by Slurm.

The pseudo code for *sigsleep* is given in Algorithm 3.1 and its source code in Appendix C. Sigsleep starts with the declaration of structures for storing eventual suspension periods and the characteristics of the job. The function *sighandler*, lines 11 to 16, is registered by *sigsleep* to handle the signal *SIGTSTP* (SIGnal Terminal SToP). This signal must be handled by a program, otherwise, it will be suspended, which is equivalent to typing *CTRL- Z* in a terminal. Each time *sighandler* is called, it obtains the current suspension time and resets the program to its default behaviour by using the function *signal(signum,SIG_DFL)*. *SIG_DFL* is a macro that expands into an integral expression that is not equal to an address of any function. The net result of calling *signal(signum,SIG_DFL)* is to reset the calling program to its default behaviour, i.e. being able to be suspended. Lastly, *sighandler()* re-sends the signal *SIGTSTP* to finally suspend itself.

Sigsleep, lines 17 to 32, starts by storing its id, priority, number of cores, submission, start and runtime in a *job_data* struct. It then enters in a *do-while* loop controlled by the function *nanosleep()*. Inside this loop, the function *sighanler()* is registered and the value of the number of suspensions incremented. In the first passage, this number will be zero and, therefore, the instructions inside of the *if* statement in line 26 won't be executed. *Nanosleep(runtime, remain)* suspends the execution of sigsleep until the time specified by *runtime* is elapsed or until a signal *SIGTSTP* is received. In the last case, the function *sighandler* will be called and the program will stop until it receives a continue (*SIGCONT*) signal. Once this signal is received, nanosleep() will return *-1* and the remaining sleeping time will be given in the parameter *remain*. This will force the program to perform a new iteration of the loop, increasing again the number of suspensions. The condition for the *if* statement will be true this time, and the resume time will be obtained and stored in a simple linked list. This loop will continue until the program is allowed to sleep without any interruptions. The program finishes after writing in a single log file (*LOG_FILE*) all data related to the job. The structure of the log file is a sequence of *job_data* structs interleaved by eventual *suspend_period* structs. The field *n_suspensions* in *job_data* specifies the number of *suspend_period* structs that follow that particular struct.

3.3.2 RunWorkload

The submission of an individual job in a Slurm cluster can be accomplished by the use of the *sbatch* program and a job description file. The main information contained in this file is the number of cores, nodes, priority and the program to be run.

A *bash* script, called *runworkload*, was developed to automate the task of submitting all jobs contained in a workload. Additional parameters were also introduced to enable studies on runtime shrinkage. Beyond the workload filename, the user can also provide the initial and final shrinking factors and a step. For instance, typing *runworkload work_file 100 50 10* in the terminal will result in 5 output files. The first output file will contain the results of the emulation with 100% of the expecting and

Algorithm 3.2: A script for workload emulation.

```
1  Function sleep_submission_period(j, trace_ini)
       Input  : Two nonnegative integers, submission period and the start of the
                trace
       Output : sleeping for regular intervals
2      sdj ← trace_ini + warmup_period + (drift + submission_period) * j;
3      sleep(sdj − now);
4  Program runWorkload(w_file,ini, end, step)
       Input  : 1 string (workload filename), 3 integers (initial, end and step
                shrink factors)
       Output : A file containing the trace emulation
5      for i ← ini to end by step do
6          n_group, n_jobs ← 0;
7          trace_ini ← now;
8          foreach job in w_file do
9              n_jobs++;
10             shrink job's submission time, expected runtime and runtime by i;
11             if (n_jobs >= n_warm_up) and (n_jobs % sub_group == 0) then
12                 sleep_submission_period(n_group, trace_ini) ;
13                 n_group++;
14             write submission script, job_description, with job info;
15             submit job (sbatch  job_description);
```

actual running times as described in the file *work_file*. These times will be decreased by steps of 10% until the final shrinkage factor of 50% in the subsequent output files.

Runworkload, presented in Algorithm 3.2, starts by submitting a preconfigured number of jobs as warm up jobs (*n_warm_up*). Once the warm up period (*warmup_period*) finishes, the following jobs are then submitted in groups (*sub_group*) in regular intervals (*sub_period*). The objective of the function *sleep_submission_period()* is to guarantee that the start of a new submission period is done in terms of wall time instead of a period relative to the last submission. This is necessary since the submission time for each job, line 15, is arbitrary and depends on how busy Slurm is. The importance of setting the sleeping periods in this way and the necessity of the *drift* parameter will be evidenced during the discussion of the emulation methodology presented in Section 4.2.4.

3.3.3 CalcMets

Calcmets, Algorithm 3.3, is a program that evaluates the classical scheduling metrics, i.e. average waiting time, response time, slowdown, weigthed slowdown and cluster utilization. The metrics are calculated from the trace data generated by the running of several *sigsleep* programs. This trace is a binary file composed by a sequence of *job_data* structs (1-4) interleaved by sequences of *suspend_period* structs (5-9).

The logic of this program is straightforward. The information about the jobs are stored in an array of *job_data* structs (16). All data concerning suspensions of a particular job is stored in a linked list accessed by the field *sus_res*. Once this information is read from the trace file, the scheduling metrics are evaluated according to the formulas presented in section 2.4.

3.4 Software platform

All programs were compiled using the GNU Compiler Collection (GCC). Programs using the *clock_gettime* function were compiled with the flags *-lrt*.

The versions of all software used in this thesis are summarized in table 3.2.

3.5 Summary

This chapter introduced the steps taken to build the computing cluster where the experiments presented in this thesis were performed. We also discussed the algorithms corresponding to three key programs: a sleeping program responsive to suspensions signals, a script to automate job submission, and a program to evaluate classical scheduling metrics. In the next chapter, the reasons for the methodology proposed in this thesis will be presented and discussed.

Algorithm 3.3: An algorithm for classical scheduling metrics evaluation

```
1  struct {
2      double suspend, resume;
3      suspend_period *next;
4  } suspend_period;
5  struct {
6      int id, n_cores, n_suspensions;
7      timespec submission, start, end, runtime;
8      suspend_period *sus_res;
9  } job_data;
10 Function read_jobs(trace, jobs)
       Input  : trace filename
       Output : array of job_data
11     while not end of trace do
12         read job data and store in jobs[i++];
13         for j ← 1 to jobs[i].n_suspensions do
14             read and add to job[i].sus_res a new suspend_period node;
15 Program calcmets(trace)
       Input  : Trace File Name
       Output : Classical Scheduling Metrics
16     job_data jobs[];
17     read_jobs(trace, jobs);
18     evaluate classical metrics;
```

Table 3.2: Software Platforms

Software	Version
GCC	4.4.7
Python	2.6.6
bash	4.1.2
OpenStack	2.28.1
Slurm	16.05.6
Munge	0.5.10

Chapter 4

Experimental Methodology

This chapter characterizes the experimental workload and details the methodology used in the emulations presented in this thesis.

4.1 Experimental Workload Characterization

We used a workload containing 4,417,018 jobs submitted to NCI's supercomputer between April 1, 2016, and November 1, 2016. This workload contains, among other information, the submission time, number of requested cores, and estimated and actual runtimes for each job. Figure 4.1 shows the distribution of the number of jobs per CPU count in a logarithmic scale.

One particular feature of this workload is the great presence of one-core jobs, which make up almost 50% of all jobs submitted during the period. This is a characteristic of NCI which not only provides HPC services, but also provides 8 petabytes of high-performance storage services. Most of the access to the data is made by one-core jobs. Table 4.1 displays the most common job sizes and their respective percentage of the total number of jobs. It is worth noting that, out of 193 different job sizes presented in the workload, the number of jobs requesting 1, 2, 4, 8, 16, 48 and 64 cores accounts for 90% of all jobs submitted.

This job size distribution could lead to the wrong conclusion that most of the work done at NCI do not actually need a supercomputer. Nevertheless, Table 4.2 shows

Table 4.1: Job size and its percentage of total jobs

Job Size (cores)	% of Total Jobs (%)
1	49
2	11
4	5
8	7
16	8
48	4
64	6
all rest	10

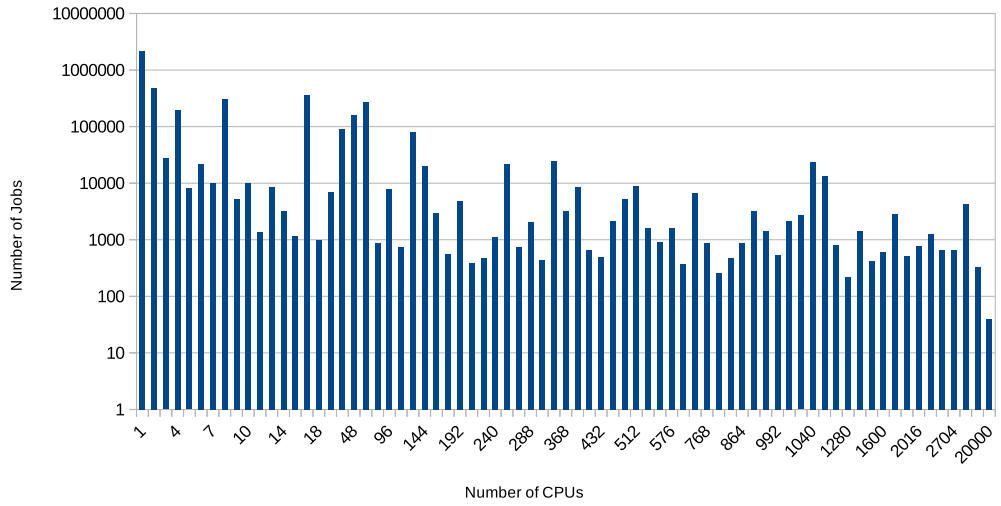

Figure 4.1: Raijin's job size distribution (Y axis is logarithmic)

the job distribution in terms of service units (S.U.), which is the runtime of a job multiplied by the number of cores used. This data demonstrates that most of the work done at NCI's supercomputer is performed by jobs that use a considerable number of cores. One core jobs, on the other hand, use only 1.6% of NCI's computational power.

One interesting feature of this workload is the distribution of the job's submission time along the day. It would be reasonable to expect a peak during working hours, yet Figure 4.2 shows a fairly uniform distribution.

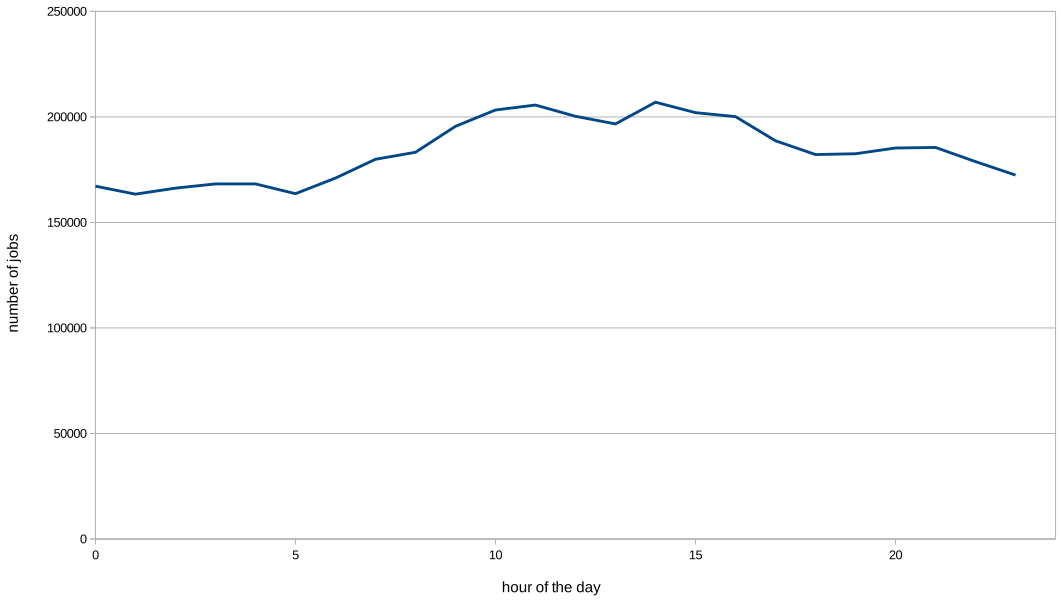

Figure 4.2: Raijin's job submission time distribution.

Table 4.2: Job size and its percentage of total Service Units (S.U.)

Job Size (cores)	Runtime (10^6 s)	S.U. (10^9 cores*s)	% of Total S.U.
64	2416.1	154.6	16.0
128	850.1	108.8	11.2
320	188.4	60.3	6.3
256	235.0	60.2	6.2
16	3154.5	50.5	5.2
512	82.9	42.6	4.4
8	5243.2	41.9	4.3
32	915.7	29.3	3.0
48	485.7	23.3	2.4
384	60.5	23.2	2.4
1008	17.7	17.8	1.9
1024	17.1	17.5	1.8
192	82.0	15.8	1.6
1	15355.0	15.3	1.6
3200	4.5	14.6	1.5
all rest			30.2

As already explained in Section 2.3, a scheduler using the Backfill algorithm demands an estimate for the runtime of a job at its submission time. Figure 4.3 exemplifies the users' notorious behaviour of largely overestimating the runtime values. This graph shows the percentage of the total jobs for each interval of the ratio between the actual runtime (RT) and the runtime estimate (ER). For instance, almost 60% of all jobs submitted have their runtime estimates more or equal than 10 times the actual runtimes, which is the first interval (0.0,0.1] showed in Figure 4.3. On the other end, only 0.7% have this ratio falling within the (0.9,1.0] interval which could be considered a very good estimate. This behaviour is easily explained by the fact that overestimating the running time incurs no penalty, whilst users might have their jobs killed if their actual runtime exceeds its runtime estimate. Usually, a job will not be killed as soon as this situation occurs but after a grace period, which explains the (1.0,1.1] interval.

4.2 Classical Metrics Artefacts

This section discusses some artefacts that affect the classical metrics. An artefact is a result arising from a scientific investigation or experiment that is not naturally present but occurs as a result of the preparative or investigative procedure. This section describes the effects of concurrency, submission ordering, sampling and time shrinking on the classical metrics. All the experiments were performed using the cluster described in Section 3.1. The most relevant parameters of the Slurm configuration file are shown in the Appendix B. Two smaller workloads, derived from the original workload described in Section 4.1, were used in the following studies and the procedures to obtain them are shown in Sections 4.2.3 and 4.2.4.

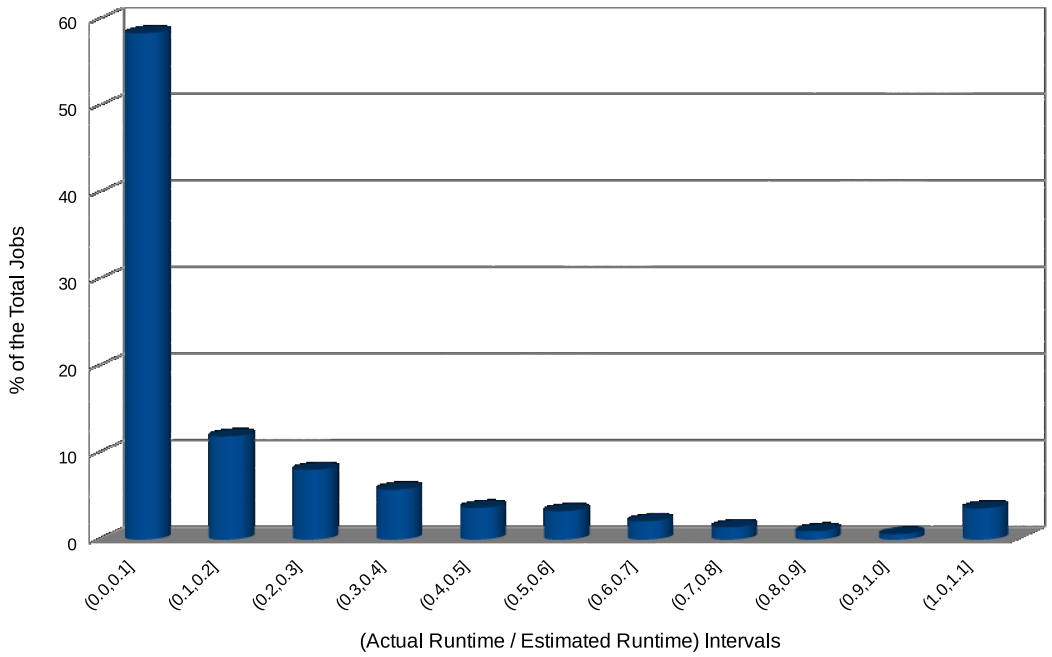

Figure 4.3: Users' accuracy for their job runtime estimates. X-axis contains the intervals for the ratio between the actual runtime and estimated runtime. Y-axis shows the percentage of these jobs in relation to the total number of jobs in the workload. For instance, all jobs with a runtime estimate equal or bigger than 10 times the actual runtime are grouped in the (0.0,0.1] interval which represents 58% of all submitted jobs.

4.2.1 Concurrency

Table 4.3: Stability of the classical metrics over the course of 5 consecutive runs. The columns are Waiting Time (WT), Response Time (RT), Slowdown (SD), Weighted Slowdown (WS) and Utilization (UT). The last 4 rows are the Average (AVE), Standard Deviation (SSD), Relative Standard Deviation (RSD) and the Confidence Interval with a level of Confidence of 95% (C95) . This notation will be repeated in the following tables.

Run	WT (s)	RT (s)	SD	WS	UT (%)
1	1261.1	1389.3	64.8	103.5	90.2
2	1286.8	1415.0	67.4	115.3	92.2
3	1256.0	1384.1	67.7	106.8	91.3
4	1294.7	1422.8	67.7	112.2	90.7
5	1304.5	1432.6	64.9	109.9	91.5
AVE	1280.6	1408.8	66.5	109.5	91.2
SSD	48.9	18.9	1.4	4.1	0.7
RSD (%)	1.5	1.3	2.0	3.8	0.8
C95	60.7	23.5	1.7	5.1	0.9

All the experiments performed in this thesis involved a reasonable number of programs running concurrently in a cluster. As expected, the values of the classical

metrics varied from one experiment to the next even when all parameters were kept the same due to the non-deterministic nature of time-dependent experiments in distributed systems. This shows that emulation presents relevant additional features when compared with simulation, which usually displays a deterministic behaviour.

Tables 4.3, 4.4, 4.5, and 4.6 utilize a 1,000 job workload running in a 3,000 node/3,000 CPU cluster. Table 4.3 presents the results for five consecutive experiments with all the cluster configuration parameters kept constant. All the metrics display a low Relative Standard Deviation (RSD). The low values for the RSD and the similar values between the standard deviation (SSD) and the confidential interval with a level of confidence of 95% (C95) indicates that these metrics are sufficiently stable in relation to concurrency to be used in comparisons to different scheduling algorithms. A particular result shown in this table is the high stability of the cluster utilization metric, which displays a RSD of only 0.8%. As we will show later, the utilization also displays a very stable behaviour in relation to all the other artefacts studied.

4.2.2 Submission order

As discussed in [Frachtenberg and Feitelson, 2005], workloads from different HPC facilities can be quite different from each other. These differences can influence the performance of the scheduling algorithms as it was found in [Mu'alem and Feitelson, 2001], where the relative performance of EASY and Conservative Backfill depended on the workload used and on the performance metrics.

This motivated us to investigate the behaviour of the classical metrics in relation to the submission order of the jobs for a particular workload. From a single workload, we generate 10 different workloads by randomly changing the submission order of the jobs. We kept the submission times equally distributed along the trace duration. This was done for two reasons: First, Figure 4.2 shows a very uniform submission time distribution over a 24 hours period while the other reason is related with the differences between a simulation and the actual working of a real RMS. In a simulation, the scheduler can run the scheduling algorithm as soon as a new job arrives. However, there are many more tasks, beyond scheduling, that a real RMS must perform. This imposes an empirical limit to the number of jobs in the queue that can be scheduled as soon as a running job completes. For instance, Slurm performs two types of scheduling, one fast that only takes into account a small percentage of the whole queue and one more comprehensive that involves the whole queue, but that is done only at preconfigured intervals. The Slurm documentation recommends disabling the fast scheduling for high throughput clusters, which can be easily accomplished by setting a parameter, *defer*, in the configuration file. We use this high throughput configuration in our cluster because of the time shrinking experiments that will be discussed in detail in the next section. An arriving job, therefore, will not trigger the scheduling algorithm in our experiments. In this case, this arriving job will be put in the end of the queue and it will be included in the scheduling process only when the scheduling period starts. In the following experiments, we used a scheduling period

of 30 seconds.

Table 4.4: Classical Metrics for a workload shuffled 10 times. All jobs had the same priority and they were scheduled with Backfill.

Run	WT (s)	RT (s)	SD	WS	UT (%)
1	1350.3	1478.4	73.4	109.7	92.0
2	1325.1	1453.2	67.2	108.7	86.6
3	1121.6	1249.7	61.9	123.7	88.1
4	1611.9	1740.0	77.7	102.4	85.6
5	1582.0	1710.0	72.6	119.1	82.2
6	1410.5	1538.6	79.7	102.2	82.2
7	1113.6	1241.7	56.7	81.2	84.4
8	1593.8	1721.9	93.9	146.3	83.7
9	1290.2	1418.3	62.3	95.5	82.9
10	1756.0	1884.2	88.2	142.7	84.2
AVE	1415.5	1543.6	73.4	113.2	85.2
SSD	204.9	204.9	11.3	19.3	2.9
RSD(%)	14.5	13.3	15.4	17.0	3.4
C95	146.5	146.5	8.1	13.8	2.1

Table 4.4 shows the effect of the job submission order on the classical metrics using backfill. The resulting traces show a RSD between 3.4% and 17.0% for the classical metrics.

The high variation of these metrics indicates that the submission order plays an important role in their evaluation. It is worth noting, however, the low value for the cluster utilization, 3.4%, and the relative high values for the other metrics. These values were consistent through all our experiments.

Next, we studied the effects of prioritization in the same workloads for two cases and two scheduling algorithms, Backfill and Suspend/Resume. Prioritization is necessary since Suspend/Resume algorithm uses it to decide what job suspend.

For each workload in Table 4.4 we randomly assign 20% of the jobs a high priority and kept the remaining jobs with low priority. In the second case, only jobs requesting 700 CPUs or more received high priority. We processed these two sets of workloads using backfill and suspend/resume. The results are summarized in Tables 4.5 and 4.6. The submission order in these new workloads was kept the same as in Table 4.4, so the effects of the introduction of priorities in the scheduling metrics and the use of different scheduling algorithms can be readily compared.

The results show that the introduction of priorities worsens the averaged values for almost all metrics. This effect can be explained since high priority jobs will be scheduled first, which constrains the number of jobs that the scheduler can use to optimize the scheduling.

Taking into account only the average values for the classical scheduling metrics, we found out that for Waiting Time (WT) and Response Time (RT), Backfill performed

Table 4.5: Classical Metrics for the same workloads and algorithm (Backfill) described in Table 4.4. The columns labeled *Random Priorities* had 20% of their jobs randomly assigned with high priority and the remaining jobs with low priority. The columns labeled *Size Based Priorities* had only jobs requesting 700 or more CPUs assigned with high priority.

Run	Random Priorities					Size Based Priorities				
	WT(s)	RT(s)	SD	WS	UT(%)	WT(s)	RT(s)	SD	WS	UT(%)
1	1340.1	1468.2	67.9	134.4	87.9	1418.5	1546.5	60.0	76.4	83.2
2	1344.9	1473.0	64.7	143.6	83.2	2536.5	2664.5	108.6	123.6	87.6
3	1198.3	1326.4	60.9	137.9	91.5	1479.9	1607.9	61.1	105.2	90.7
4	1418.0	1546.2	74.9	140.2	84.7	2118.3	2246.3	89.0	94.0	88.2
5	1704.0	1832.1	81.5	153.2	80.7	2247.9	2375.9	90.8	105.1	81.3
6	1400.1	1528.2	73.1	134.7	89.2	1914.8	2042.8	82.7	92.3	88.3
7	1382.7	1510.8	65.9	120.6	86.1	1675.3	1803.3	68.5	73.4	88.5
8	1457.5	1585.6	72.5	140.1	90.5	2200.8	2328.8	101.1	124.2	84.8
9	1277.6	1405.7	62.3	166.3	83.8	1601.7	1729.7	60.9	86.9	88.6
10	1979.4	2107.6	91.6	167.5	89.5	2564.4	2692.4	108.1	143.1	95.1
AVE	1450.3	1578.4	71.5	143.9	86.7	1975.8	2103.8	83.1	102.4	87.6
SSD	216.6	216.6	9.0	13.9	3.4	399.5	399.5	18.5	21.3	3.7
RSD (%)	14.9	13.7	12.6	9.7	3.9	20.2	19.0	22.3	20.8	4.2
C95	154.9	154.9	6.4	9.9	2.4	285.8	285.8	13.2	15.2	2.6

Table 4.6: Classical Metrics for the same configuration as in Table 4.5. The scheduling algorithm used was suspend/resume.

Run	Random Priorities					Size Based Priorities				
	WT(s)	RT(s)	SD	WS	UT(%)	WT(s)	RT(s)	SD	WS	UT(%)
1	1704.3	1953.5	81.9	124.8	89.4	2423.1	2623.4	111.2	104.7	82.9
2	1465.8	1678.5	65.0	121.8	84.0	2963.8	3112.8	130.4	117.4	87.3
3	1920.4	2125.2	87.7	143.0	78.2	2523.0	2709.6	113.8	123.0	88.3
4	1438.7	1635.6	68.1	117.9	92.3	2140.6	2346.3	88.6	88.2	88.0
5	1404.6	1627.0	67.3	115.8	95.7	2546.3	2700.0	109.5	114.4	82.3
6	1830.4	2052.6	89.0	110.2	93.0	2269.0	2456.6	101.6	100.9	88.4
7	1640.4	1865.6	79.3	115.6	88.0	2179.4	2352.1	99.8	88.4	88.6
8	1915.4	2116.0	94.1	159.7	81.3	2656.8	2899.8	124.1	119.0	84.6
9	1627.5	1860.1	77.7	129.7	87.9	2347.1	2499.1	95.6	108.4	89.3
10	2042.4	2188.8	90.2	149.0	87.8	2659.8	2790.2	115.7	146.9	95.9
AVE	1699.0	1910.3	80.0	128.8	87.8	2470.9	2649.0	109.0	111.1	87.6
SSD	211.8	200.5	9.9	15.6	5.1	239.6	233.3	12.2	16.5	3.7
RSD (%)	12.5	10.5	12.4	12.1	5.8	9.7	8.8	11.2	14.9	4.2
C95	151.5	143.4	7.1	11.2	3.7	171.4	166.9	8.7	11.8	2.6

better than suspend/resume as can be summarized in the following relationship

$$\text{Backfill} < \text{Backfill (rand)} < \text{SR (rand)} < \text{Backfill (size)} < \text{SR (size)} \quad (4.1)$$

where *SR* stands for Suspend/Resume, *rand* refers to the experiments with randomize high priorities given to 20% of the jobs, and *size* refers to experiments with high priorities given to jobs asking for 700 or more cores.

Slowdown, on the other hand, displayed a slightly different behaviour. We observed that Backfill with random assigned priorities showed a lower value than Backfill with no priorities. These measurements, however, displayed very close values, 73.4 and 71.5 and relative high Standard Deviation, 11.3 and 9.0, which makes difficult to draw any conclusion. Based solely on the average values, the relationship for slowdown is:

$$\text{Backfill (rand)} < \text{Backfill} < \text{SR (rand)} < \text{Backfill (size)} < \text{SR (size)} \quad (4.2)$$

Weighted slowdown, as discussed in 2.4, is a metric that favors big jobs and, as one might expect, it is also reflected in our experiments. The best performance for this metric was obtained for the workload with high priorities given to big jobs for both Backfill and Suspend/Resume algorithms:

$$\text{Backfill (size)} < \text{SR (size)} < \text{Backfill} < \text{SR (rand)} < \text{Backfill (rand)} \quad (4.3)$$

being the worst case Backfill with random priorities.

Utilization, on the other hand, did not show any clear pattern in the performed experiments, Tables 4.4 to 4.6. For some workloads Backfill gives a better value, like run number 1, others, like run number 10, favor Suspend/Resume with high priorities associated with big jobs. However, all values for this metric were very similar and within relatively low standard deviations.

Table 4.7: Classical Metrics for the workload described in Section 4.2.3 shuffled 6 times. The resulting trace was obtained using backfill as the scheduling algorithm. All jobs had the same priority.

Run	WT (s)	RT (s)	SD	WS	UT
1	1204.5	1642.4	252.2	74.2	95.1
2	1372.3	1802.0	280.4	85.2	95.0
3	1037.6	1491.3	215.3	60.6	94.5
4	1413.6	1845.0	363.4	91.7	95.2
5	1511.7	1934.6	313.4	82.9	94.9
6	1693.0	2119.0	327.0	106.1	95.9
AVE	1372.1	1805.7	292.0	83.5	95.1
SSD	209.8	200.0	50.0	10.0	0.4
RSD(%)	15.3	11.1	17.1	12.0	0.4
C95	220.2	209.9	52.5	63.6	99.2

The main objective of these experiments was to study the influence of the sub-

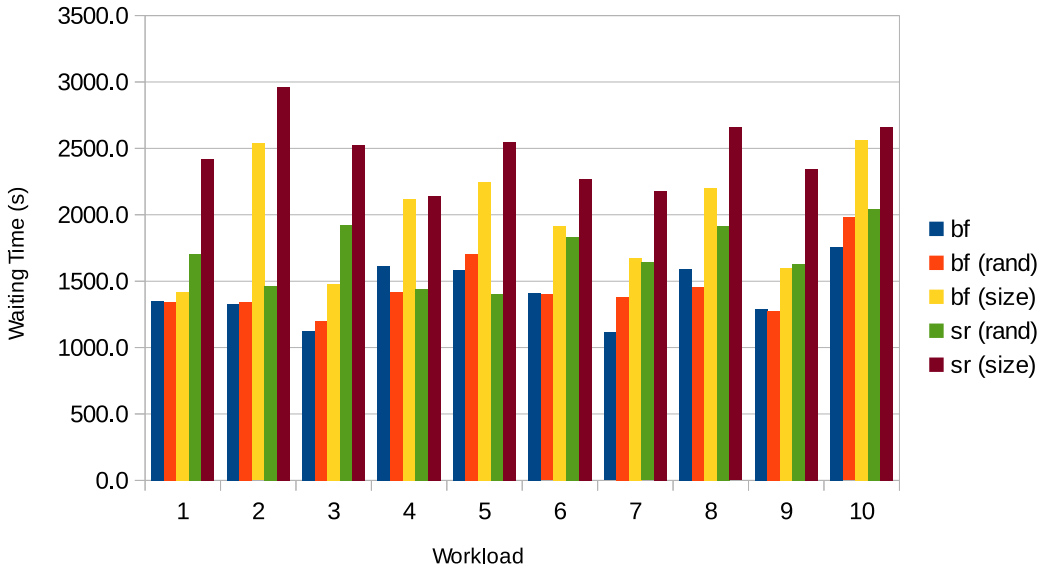

Figure 4.4: Waiting time values for a workload with the submission order shuffled 10 times. Each new workload gives rise to three more workloads. The first workload has all jobs with the same priority, the second contains 20% of the jobs randomly assigned with high priority (*rand*), and, in the third workload, high priorities are assigned only to jobs asking for 700 or more CPUs (*size*). Backfill is applied to all these workloads and the resulting Waiting Times are labeled *bf*, *bf (rand)*, and *bf (size)*. Suspend/Resume is applied only to the workloads with different priorities and the resulting Waiting Times are labeled *sr (rand)* and *sr (size)*.

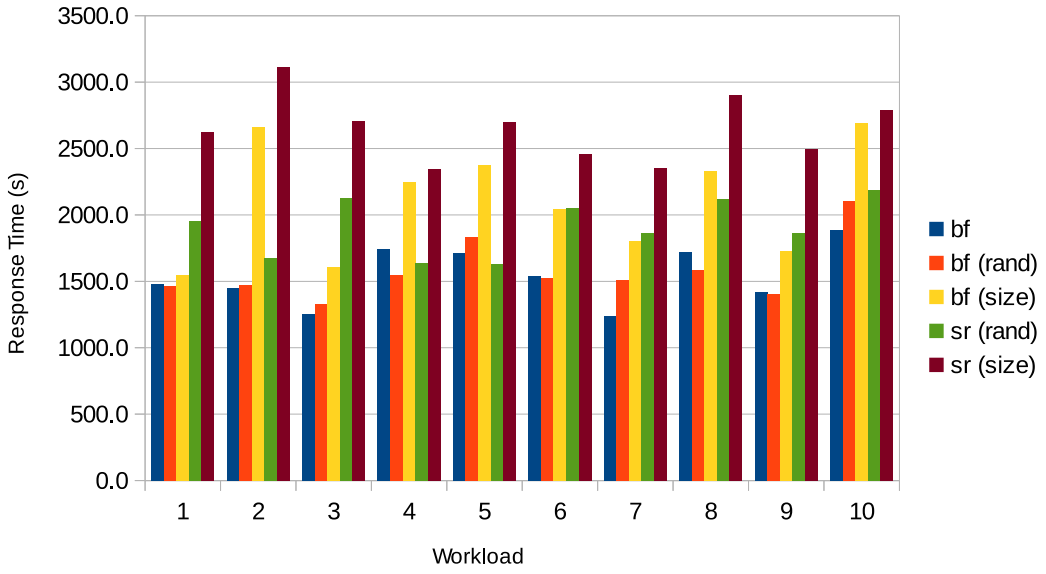

Figure 4.5: Response Time values for the same set up shown in Figure 4.4.

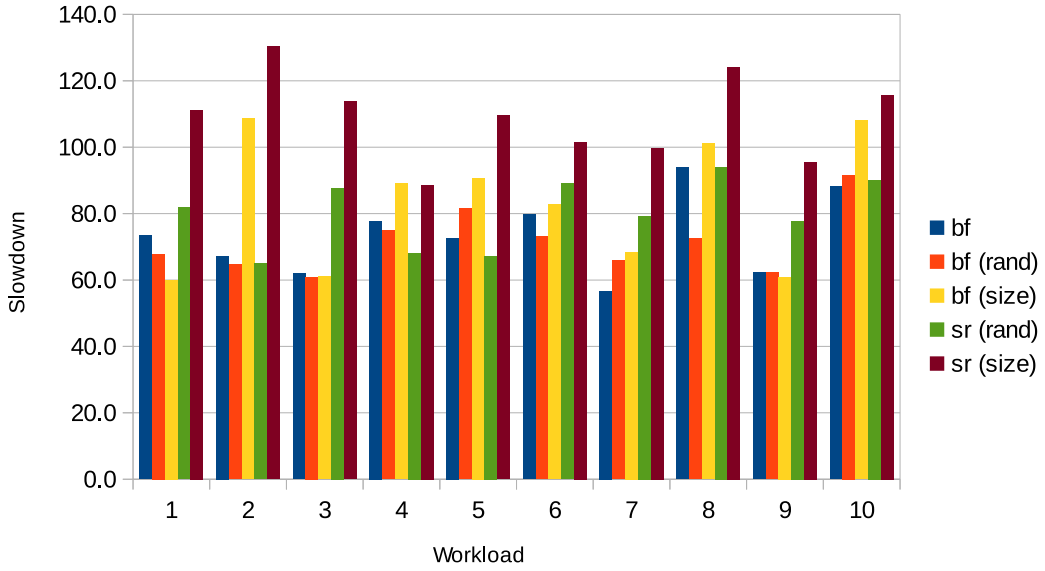

Figure 4.6: Slowdown values for the same set up shown in Figure 4.4.

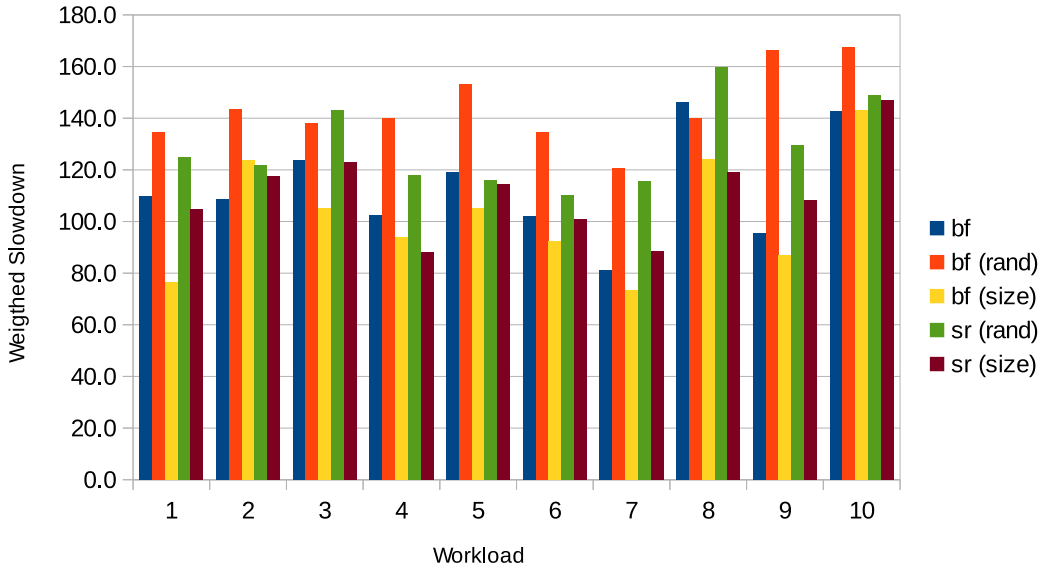

Figure 4.7: Weighted Slowdown values for the same set up shown in Figure 4.4.

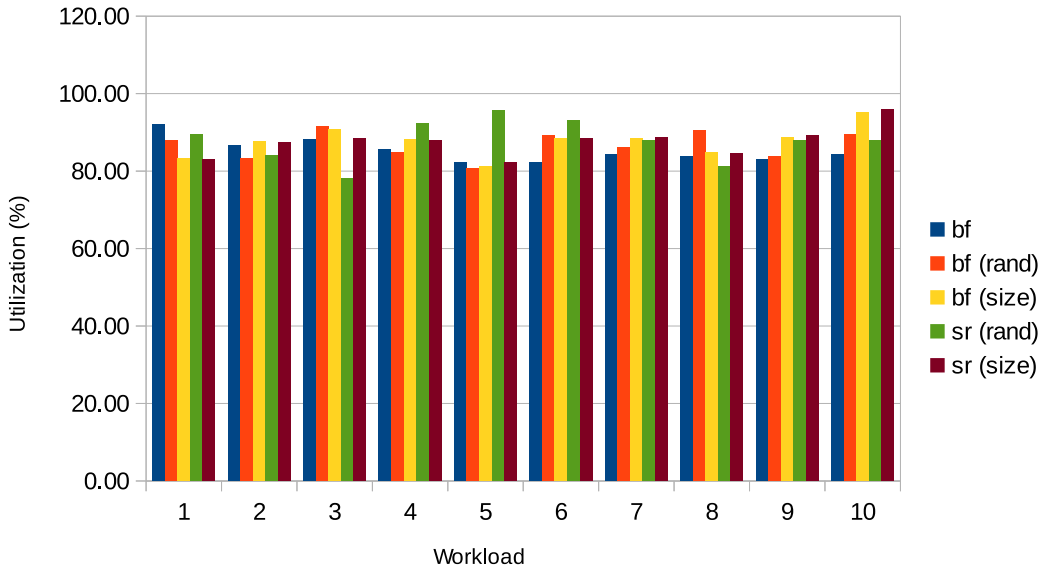

Figure 4.8: Utilization values for the same set up shown in Figure 4.4.

mission order in the values of the classical metrics. Figures 4.4, 4.5, 4.6, 4.7 and 4.8 display the behaviour of each individual classical metric here studied accordingly with the submission order, priorities and scheduling algorithm.

On average, the introduction of priorities worsens the values of the Waiting Time, Figure 4.4, as stated in the Equation 4.1. However, this is not true for all individual workloads. For instance, while workloads 1, 2, 6 and 9 display similar values for Backfill with and without priorities (blue and red bars), workloads 4 and 8, favor Backfill without priorities. The other workloads do not show significant differences. One can also not draw a definitive conclusion about what is the best scheduler algorithm in relation to this metric when analysing individual workloads. When comparing Backfill and Suspend/Resume with random priorities (red and green bars), workloads 1, 2, 3, 6, 7, 8 and 9 favors Backfill, workload 5 favors Suspend/Resume, while the remaining workloads give similar results. When high priorities are given only for big jobs a clearer scenario emerges (yellow and brown bars). In this case, practically all workloads give better results for Backfill, except for workload 4 which displays similar values.

Figures 4.4 and 4.5 show practically identical behaviour for the Waiting time and Response Time metrics. This can be explained due to the similarities in their definitions, Equations 2.3 and 2.4. A simple mathematical manipulation can easily show that the Response Time is equal to the Waiting Time plus the average running time of all jobs.

Slowdown, Figure 4.6, also displays a sensitive behaviour to the submission order. The Suspend/Resume algorithm applied to the workload with size based priority still provides the overall worst results. Backfill with random priorities (red bar) operates better than Suspend/Resume (green bar) in 6 workloads (1, 3, 6, 7, 8 and 9), similar

in 2 (2 and 10) and worse in 2 (4 and 5).

Weighted Slowdown, as expected, shows smaller values for the workloads that prioritize big jobs, Figure 4.7. For this metric, however, Suspend/Resume shows now better results for random priorities (green bar) for workloads 1, 2, 3, 5, 6, 9 and 10 when compared to Backfill (red bar), which is almost the opposite results found for Waiting Time, Figure 4.4.

Figure 4.8, exhibits the Utilization for all workloads and prioritization schemas. The visual inspection of this graph can confirm that this metric displays very stable values through all workloads, prioritization and the two scheduling algorithm used.

As an indication that this behaviour is not a feature of this workload and cluster size, Table 4.7 compiles the results of a bigger workload with 30,000 jobs processed in an also bigger emulated cluster with 57,600 CPUs. This workload was shuffled 6 times and the classical scheduling metrics were evaluated in the same way shown in the previous tables. The values for SSD and RSD are compatible with the values found in the previous tables which indicates that the submission order is an artefact for the classical metrics.

We conclude this section by emphasizing the importance of the submission order in the evaluation of the classical metrics. The results of this section show that a particular order of the job submission can favors an algorithm conveying lower values for the classical metrics. We suggest, therefore, that a simpler and more effective way of overcoming this artefact is through the use of shuffling and averages.

4.2.3 Sampling

Workloads from HPC facilities usually contain thousands or even millions of jobs and cover long periods. The original workload from NCI's supercomputer used in thesis contains 4,417,018 jobs and spans a period of 214 days. Since it is impractical to work with emulations for such a long period, strategies must be developed to reduce the emulation time. We use two strategies to make the emulation more time feasible, sampling and time shrinking. The objective of sampling is to obtain a smaller subset of the original workload with the same characteristics and that renders similar values for the classical metrics. We will refer to this smaller subset of the original workload as the working workload. Time shrinking has the same objective and will be discussed in detail in the next subsection.

Several strategies can be devised for sampling a big workload. The simplest would be to arbitrarily choose a day from the workload, which would only yield a reasonable result if the workload had a uniform distribution of sizes and runtimes along its whole duration. Figure 4.9 displays the daily job submission for the 4 most popular job sizes for the workload described in Section 4.1. As can be easily seen from this graph, the number of jobs of any particular size changes considerably on a daily basis. This is not a particularity of NCI but a common trend among other HPC facilities as was noted by other researchers, [Niemi and Hameri, 2012], which have suggested that adaptive scheduling strategies should be adopted to deal with this variability.

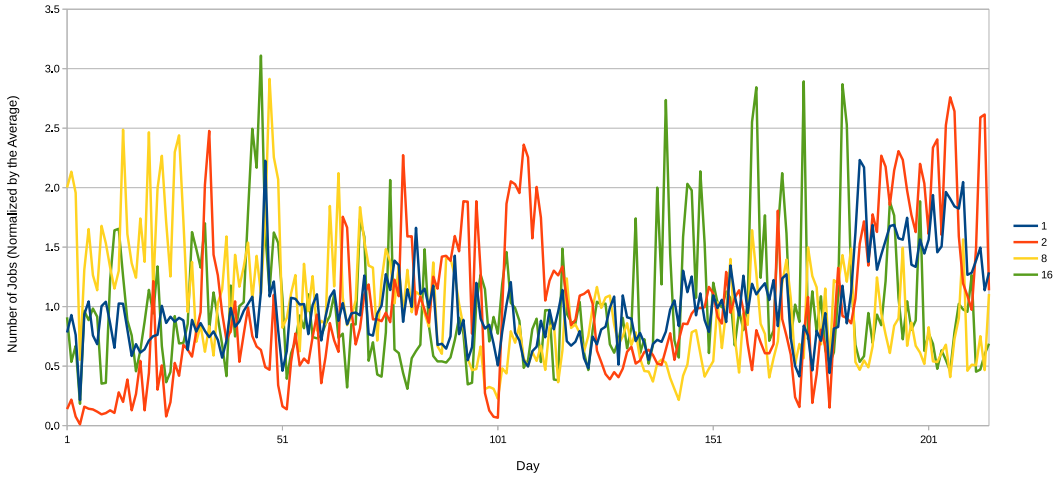

Figure 4.9: Raijin's job daily submission distribution for the 4 most popular job sizes normalized by their average. 1 core jobs displayed an average of 10,028 jobs/day, 2 core jobs 2,220 jobs/day, 8 core jobs 1,423 jobs/day, and 16 core jobs 1,715 jobs/day.

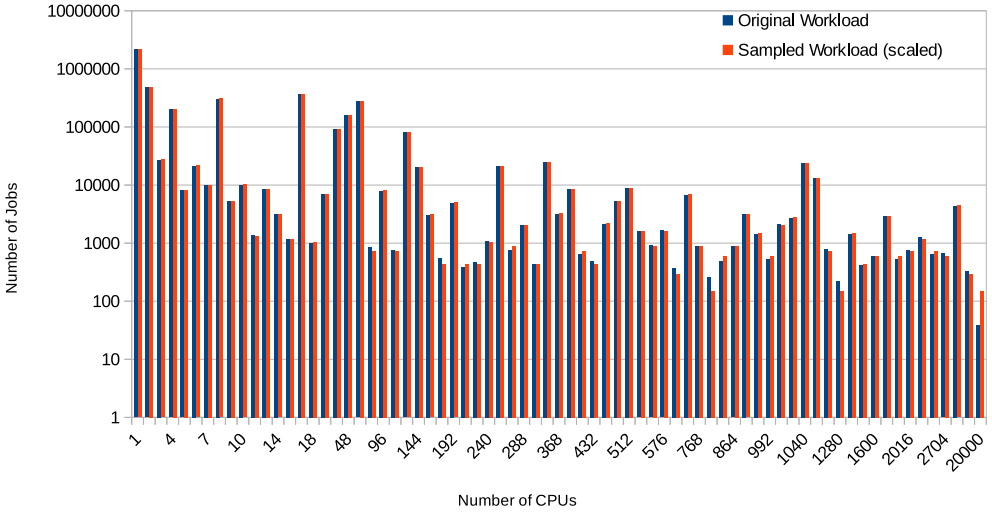

Figure 4.10: Comparison between original and sampled workload (scaled) job size distribution (Y axis is logarithmic). The scale factor (147.2) is the number of jobs in the original workload (4,417,018) divided by the number of jobs in the sampled workload (30,000).

Another strategy would be randomly select jobs from the workload, but this also wouldn't guarantee a workload with a similar size distribution of the original workload.

A technique to obtain a working workload that reflects the size distribution of the whole original workload period could be:

1. Successively order the original workload by the required runtime, runtime and size of each job.

2. Select every job_i in the ordered workload given by

$$job_i = \lfloor s_p \, i \rfloor + o_f \tag{4.4}$$

where s_p is the sampling ratio and o_f is an offset value between 0 and s_p, which determines what jobs to choose within the sample population. The sampling ratio is given by

$$s_p = \frac{N_{orig}}{N_{work}} \tag{4.5}$$

where N_{orig} is the number of jobs in the original workload and N_{work} is the number of desired jobs in the working workload.

3. Re-order the jobs in the working workload using the jobs position in the original workload.

Figure 4.10 displays a comparison between the size distribution of the original workload and the scaled size distribution of a 30,000 job working workload. The similarity between the size distributions indicates that the proposed methodology achieves the desired result.

A 30,000 job working workload implies a sampling ratio of approximately 147.2 (4,417,018 / 30,000), which means that, on average, we choose 1 and discard 146 other jobs from the original workload.

In order to estimate the impact of this sampling, we prepared 10 workloads with the same sampling ratio, but with 10 equally spaced offsets o_f. The values of the classical metrics obtained from these workloads are shown in Table 4.8. The comparison between the values of the *RSD* found in this table and the same values displayed in Table 4.7, shows that this sampling introduces an even bigger variation in the classical metrics than the shuffling of the submission order. A possible explanation for this increase is that two jobs obtained from the same sampling ratio but different offsets $o'_f s$ will have, in most cases, the same size but possible different submission times and runtimes. The added effects of shuffling and different runtimes could explain the increase in the variation of the classical metrics shown in Table 4.8.

Next we proposed a improved sampling technique. We based this technique on the assumption that similar jobs, i.e., comparable sizes, required runtime and runtimes, and submitted at similar times (having similar positions in the workload), will equally contribute to the classical metrics. In order to group these jobs we:

1. Successively order the original workload by the runtime, required runtime and size of each job. Notice that we now sort first the runtime.

Table 4.8: Classical metrics for 10 different workloads with a sampling ratio of 147.2, but with different offsets using Backfill.

Offset(o_f)	WT (s)	RT (s)	SD	WS	UT
0	666.3	1638.8	94.0	42.1	97.1
14	1048.3	1790.8	147.6	62.4	97.0
28	995.3	1749.5	138.6	63.0	97.3
42	1366.8	1915.3	190.0	78.3	97.4
56	2003.0	2163.3	280.1	116.7	97.3
70	943.5	1752.0	131.4	59.6	97.4
84	1270.5	1890.1	178.2	75.9	97.1
98	1229.5	1853.8	172.3	80.5	96.9
112	1644.5	2012.3	232.6	102.3	95.6
126	1678.8	2046.5	233.9	98.1	96.8
AVE	1284.6	1881.2	179.9	77.9	97.0
SSD	380.7	150.7	53.3	21.4	0.5
RSD(%)	29.6	8.0	29.6	27.5	0.5
C95	272.3	107.8	38.1	15.3	0.4

2. Group all jobs with the same size and required runtime falling within a window of w seconds.

We used a time window for the required runtime to increase the number of jobs inside a group and, consequently, to increase the likehood of grouping jobs with similar size and runtime.

3. Order the jobs in each group by their position in the original workload.

We expected now to have in each group jobs with similar overall characteristics.

4. Evaluate the average service units (size times runtime) of the first s_p jobs.

5. Select the job which service unit has the closest value to the average.

6. Compute the difference between the service unit of the selected job and the average of the s_p jobs.

7. Evaluate the average service units of the next s_p jobs but adding the difference calculated in the previous step.

8. Repeat this procedure from step 5 until the workload has been completely processed.

The last step 6 guarantees that the final service units in the working workload, in relation to the original workload, holds the same proportion as the sampling ratio.

Figure 4.11 shows the results for 17 sampling experiments using this improved technique with a window of 60 seconds. The sampling ratio in this graph went from 100 to 468.5. Below this sampling ratio the experiments would take too long to perform. Above the sampling ratio of 468.5 the results were inconclusive. Apart from Weighted Slowdown, the variation of the other classical metrics show that these are still affected by the sampling, but to a lesser extent than when the workload is shuffled. The still large variations on the Weight Slowdown metric could be attributed to the low number of big jobs present in the original workload. Jobs bigger than 1,000

CPUs represent only 1.2% of the total number of jobs in the original workload and the proposed sampling technique seems unable to distribute them equally in the working workloads.

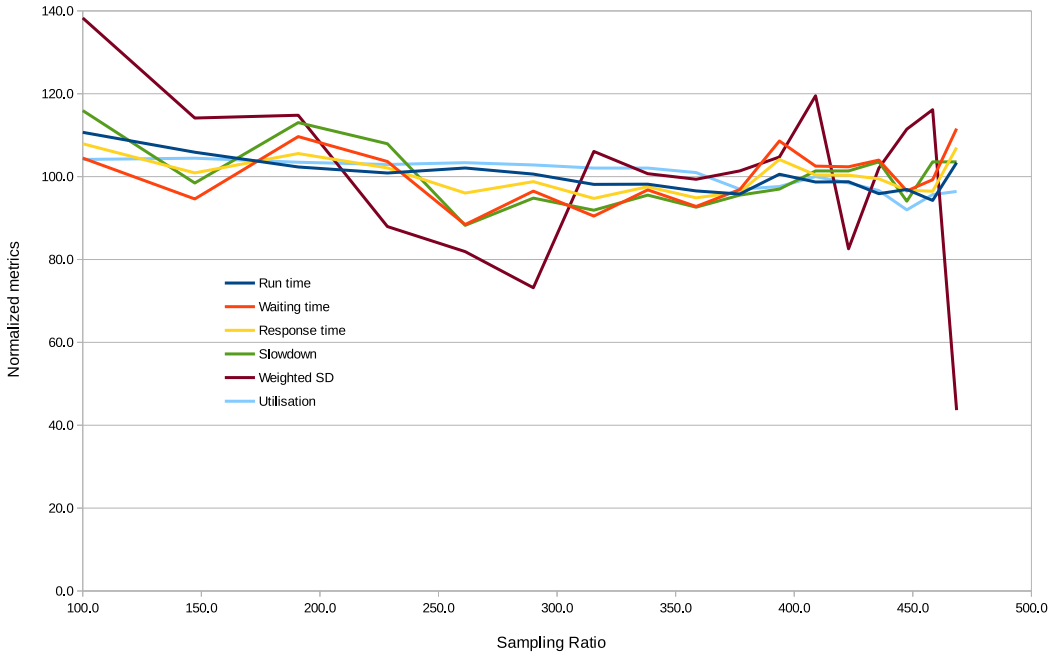

Figure 4.11: Classical metrics for 17 sampled workloads scheduled with Backfill.

4.2.4 Time Shrinking

Time shrinking is a technique for speeding up emulations. It simply decreases each time of a workload by a percentage called here the shrinking factor. This section discusses how the classical metrics behave under time shrinking and the techniques to approximate experimental and expected results.

It is straightforward to show that, if we multiply each time in the workload by a shrinking factor, some classical metrics like waiting time and response time, Equations 2.3 and 2.4, would be affected by the same amount. Other classical metrics, like slowdown and weighted slowdown, on the other hand, should not be affected since both the numerator and denominator are equally affected by the shrinking.

The key elements of our shrinking experiments are a workload file; the script for job submission, *runworkload*, described in the Section 3.3.2; a sleep program, *sigsleep*, Section 3.3.1 and a program, *calcmets*, for the evaluation of the classical metrics, Section 3.3.3. Originally, each line of the workload file contained six fields: job id, priority, submission delay, number of CPUs, expected runtime, and runtime.

A run of a shrinking experiment would have four basic steps:

1) The *runworload* script reads the workload file and, for each line, it sleeps the number of seconds determined by the submission delay field. Immediately after that,

it submits a *sigsleep* job to Slurm using the information contained in the remaining fields;

2) Just before it finishes, each *sigsleep* job writes in a common log file all the information about its run (id, submission, start and end times, runtime and number of CPUs requested;

3) After all jobs have finished, the log file is copied to another directory and reset;

4) The classical metrics are then evaluated using *calmets*.

Once these steps are completed, a new workload file is generated by decreasing all times in the original workload by a shrinking factor and the four steps aforementioned are repeated.

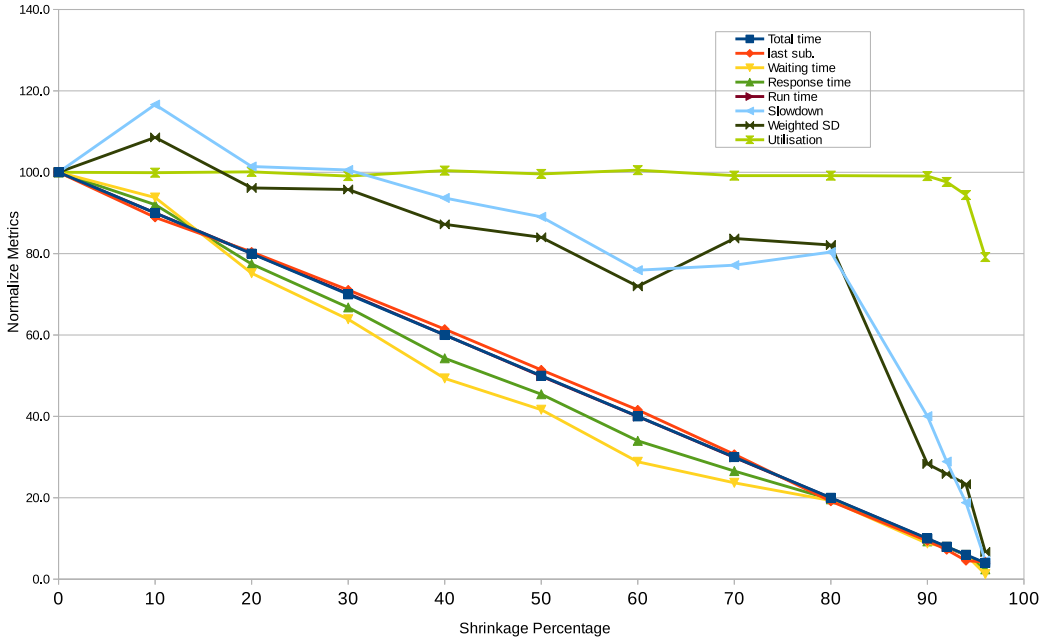

Figure 4.12: Preliminary results for time shrinking.

Figure 4.12 shows the preliminary results of our shrinking experiments. The metrics display the general expected trend where waiting and response times have a decreasing linear dependence on the shrinking factor, while slowdown and weighted slowdown should not be affected. Utilization, on the other hand, displayed a very stable behaviour until the shrinkage percentage reaches 92%, when it starts dropping considerably.

Despite the metrics displayed the expected general behaviour we decided to investigate in more depth what could be done to improve this result. Figure 4.12 also displays, beyond the classical metrics, the total time of the trace and the time of the last job submission. The trace total time is determined by the *runworkload* script as an exact percentage of the original workload total time. Therefore, this metric presents a perfect linearity with relation to the shrinkage percentage in the graph. But the time of the last submission, in a 30,000 job workload, depends on the submission times for

each of the 29999 previous jobs. Note that, in this case, the first version of the *runworkload* script submitted each job to Slurm after sleeping the submission delay time. If Slurm was busy doing something else, like scheduling, this submission would not be instantaneous and the accumulated effect of the submission of thousands of jobs would definitely affect the time of the last submission. It is also interesting noting the inverse relation between the last submission and the waiting and response times in this graph.

It should be expected that these values would lie on the same straight line as the total time. However, as the last submission displays values below this straight line, waiting and response times display values above the same. When the last submission shows values above, waiting and response time show values below. This can easily be explained as the last submission is a good indicator of the submission time of each job. If the last submission lies below the straight line it means that, on average, the jobs were submitted before their expected times, which would increase the waiting time provided each job started at its expected time. Apparently the jobs actually start at the expected time since the final utilization was constant for all shrinkage factors up to 90%. Since the jobs were submitted before but started at the correct times, the waiting and response times increased. Slowdown and weighted slowdown follows the response time, as they are directly derived from it. When the response time is above its expected value the same occurs with slowdown and weighted slowdown.

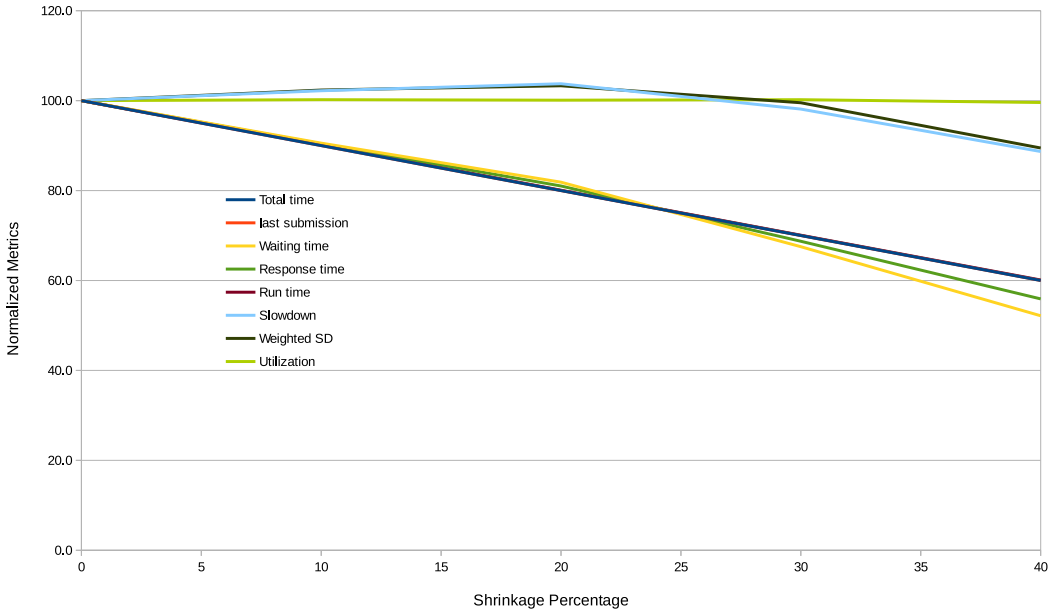

Figure 4.13: Classical Metrics behaviour using wall clock submission time.

In order to keep the submission time independent of the time of all previous submissions, *runworkload* was modified to submit jobs in terms of the wall clock time. Before starting to submit jobs, *runworkload* first obtains the current wall clock time and calculates when the next submission time is due and, only then, it submits the job.

Table 4.9: Trace with 10 equal jobs with runtime of 60 seconds being submitted at regular intervals of 60 seconds. The time delay between the ending of a job and the starting of the next induces a drift around 0.1 second per submission in the start time of each job. The accumulated effect of this drift by the 10^{th} submission is around 1 second.

ID	Sub (s)	Start (s)	End (s)	RU (s)	WT (s)	RT (s)	SD	WS
1	0.0000	0.0000	60.0002	60.0000	0.0000	60.0002	1.0000	100.0003
2	0.0000	60.0933	120.0936	60.0000	60.0933	120.0936	2.0016	200.1561
3	60.0249	120.2077	180.2080	60.0000	60.1829	120.1831	2.0031	200.3052
4	120.0049	180.3204	240.3207	60.0000	60.3155	120.3158	2.0053	200.5263
5	180.0041	240.4387	300.4388	60.0000	60.4346	120.4347	2.0072	200.7245
6	240.0047	300.5492	360.5495	60.0000	60.5445	120.5448	2.0091	200.9080
7	300.0042	360.6497	420.6499	60.0000	60.6455	120.6457	2.0108	201.0761
8	360.0038	420.7510	480.7512	60.0000	60.7472	120.7474	2.0125	201.2457
9	420.0056	480.8393	540.8396	60.0000	60.8337	120.8340	2.0139	201.3900
10	480.0042	540.9488	599.9490	59.0000	60.9446	119.9448	2.0330	203.2960

After the submission, which can take an arbitrary time due to how busy *Slurm* could be, *runworkload* obtains again the current time and calculates how long it should sleep so to wake up in the precise time for the next submission.

Figure 4.13 displays a shrinking experiment up to 40% with the above corrections. This graph shows a clear improvement when compared to the graph shown in Figure 4.12, but still displays deviations from the expected behaviour.

A new experiment was then devised to understand the possible reasons behind this behaviour. A simpler workload containing 10 equal jobs, each of them asking for 100 CPUs and running for 60 seconds, was processed by a smaller 100 CPU cluster and the resulting trace is shown in Table 4.9. Each job was submitted in regular intervals of 60 seconds and since each job asked for the total cluster capacity, they run in sequence. For the second job onwards we should have obtained the same values for the classical metrics for each job, i.e. 60 seconds for waiting time, 120 seconds for response time, 2 for slowdown and 200 for weighted slowdown. However, the time delay between the ending of a job and the starting of the next induced a drift around 0.1 second per submission. The effect of this drift continuously increased the values of the classical metrics during the trace period. For instance, the waiting time for the second submission was practically 60 seconds, while the waiting time for the last was almost 61 seconds. Since the value of this drift was constant and independent of the shrinkage factor, its effects on the classical metrics grew as the runtime decreased.

Figure 4.14 shows the classical metrics behaviour under shrinking when the submission time had its value controlled by the *runworkload* script and corrected by the drift. Now, the classical metrics displayed the expected behaviour in relation to the shrinkage factor, i.e., waiting time and response time decreased linearly, while slowdown and weighted slowdown were not affected. This relationship holds for values below a shrinkage factor of 92-94%. For shrinkage factor values above this threshold two factors negatively affect the results. First, the runtime of some jobs in our workload decreased to values less than a second, so they stayed for a very

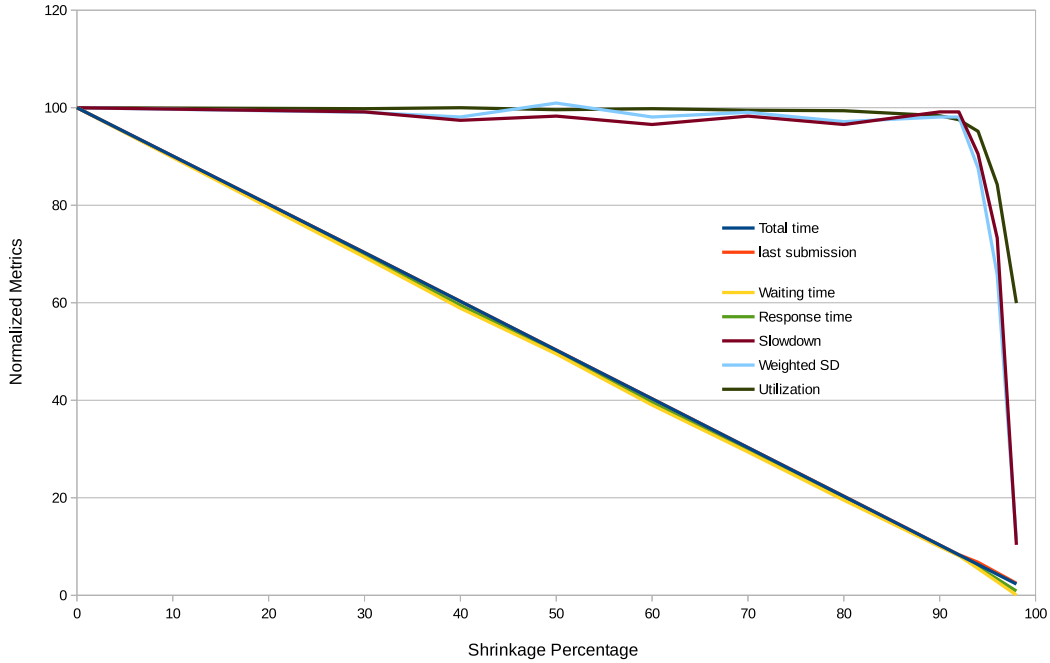

Figure 4.14: Classical Metrics behaviour using wall clock and drift corrections for a 30,000 jobs processed in a 57600-CPU cluster.

short period in the cluster. Second, the job submission frequency also grew with the shrinkage factor. These two factors combined made Slurm so busy dealing with the ending and arriving of the jobs that it could no longer schedule the workload efficiently. This caused jobs to stay longer in the queue, which led to a low utilization of the cluster and the loss of the correlation between the classical metrics and the shrinkage factor.

4.3 Summary

This chapter characterized a NCI workload. This included the job distribution in terms of size and service units, submission time, expected runtime accuracy. We also discussed some classical metrics artefacts; concurrency, submission order, sampling and time shrinking. We concluded that the submission order was an important artefact and that shuffling and averaging was a technique to overcome this problem when analysing scheduling algorithms. Sampling a workload was also a valid method to decrease the dimensions of a workload to a more manageable size. Sampling unfortunately induces, like the submission order but in a lesser degree, variations in the classical metrics. The last studied artefact was time shrinking. Time shrinking can be used to speed up the processing of a workload but additional care must be taken to keep the classical metrics linearly invariant. In this case, the submission time must be externally controlled and constant drifts must be added to each submission

period.

Chapter 5

Results

The previous chapter characterized the workload used in this thesis. It also discussed a methodology to speed up the analysis of scheduling algorithms.

This chapter compares the proposed methodology with other proposed frameworks. The methodology is then applied to compare the performance the two scheduling algorithms implemented in Slurm, i.e., Backfill and Suspend/Resume.

5.1 A Viable Environment to Evaluate Scheduling Algorithms

Supercomputers are continuously increasing in size, utilization and complexity. However, there is a lack of frameworks to support the software development lifecycle and for the comparison of different RMS packages [Rodrigo et al., 2017]. Simulation modes in the RMS packages have a notorious history of bad support [Lucero, 2011]. This behaviour is usually attributed to the costs of keeping synchronized two slightly different versions of the same software.

One of the last contributions to this field is the work of [Rodrigo et al., 2017] were the scalable Scheduling Simulator Framework (ScSF) that supports and automates workload modeling and generation, Slurm simulation, and data analysis was introduced.

This paper corroborates the relevance of the research performed in this thesis since the objectives are basically the same. However, we believe that we reach the same basic results in a simpler and more flexible way.

A key difference between the work presented in [Rodrigo et al., 2017] and this thesis is that the first still make use of a Slurm simulator, while the latter uses the latest available version of Slurm. This implies that, in their case, they will need to continually modify the Slurm simulator to keep up with the new versions of Slurm. That was the exact reason why the support for the initial versions of the Slurm simulator was dropped. We consider, therefore, our approach more appropriate since it doesn't rely on the maintenance of a complex software like the Slurm simulator.

It is also worth noting that the reported speed-up obtained by the Slurm simulator was on the average 10 times, with a maximum of 15 times. The same speed-up was obtained in our experiments by means of time shrinking. We have shown that it is possible to shrink the runtimes of a workload by 90-97% and still obtain classical

metrics that are linearly correlated to the original workload. A shrinkage interval of 90-97% corresponds to speed-ups of 10-14 times. In our case, the job submission and the job termination frequencies increased with the time shrinkage. We found that the shrinkage limit was reached when Slurm, even when configured for a high throughput cluster, could no longer have time to process job submissions, job terminations, scheduling as well as all other tasks of a RMS. Above this shrinkage percentage, CPUs were kept idle because Slurm didn't have time to schedule jobs for them, which made the utilization and consequently all other metrics to lose the linearity with the shrinkage percentage.

Still, both research relies on special features of Slurm to emulate bigger clusters. In Section 6.1, Future Work, we propose one way of overcoming these limitations with the use of containers.

5.2 Backfill and Suspend/Resume Evaluation

In this section, we compared two scheduling algorithms implemented in Slurm, Backfill and Suspend/Resume.

The nodes in a supercomputer can be utilised in two modes. In one mode, a job can only request the whole node. At a facility like NCI it would imply that any job could only request a multiple of 16 CPUs. Slurm supports this operation mode through a configuration that uses the Linear plugin. Another mode is when a job has the possibility of requesting individual CPUs. The Slurm configuration that supports this mode is called Consumable Resources. Since our workload contains jobs asking for less than 16 CPUs, we initially configured Slurm to operate with the Consumable Resources plugin.

The scheduling of big jobs usually implies a drop of a cluster's utilization when Backfill is used as the scheduling algorithm. For instance, the original NCI workload contains 14 jobs asking for 20,000 CPUs. Such big jobs would require almost 35% of the NCI's supercomputer capacity to run.

For next the experiment, we gave high priority to all jobs in our workload asking for more than 3,800 CPUs. There were 31 jobs in this category with only one of these requiring 20,000 CPUs.

Figure 5.1 shows how the cluster utilization was negatively affected by the submission of a high priority 20,000-CPU job. Because of its high priority, this job was immediately put in the beginning of the queue. As jobs finished, the newly released CPUs were not used to run other jobs but instead they were kept idle until the scheduler had collected enough CPUs to run this big job. This caused a drop in the utilization between the submission and the start of a big job, as illustrated by Figure 5.1.

The Suspend/Resume algorithm, on the other hand, relies on the immediate suspension of lower priority jobs to run a high priority job. However, as shown in Figure 5.2, we detected an anomalous drop in the cluster utilization when big jobs are submitted. Figure 5.3 shows that the reason for this drop is that Slurm is suspending

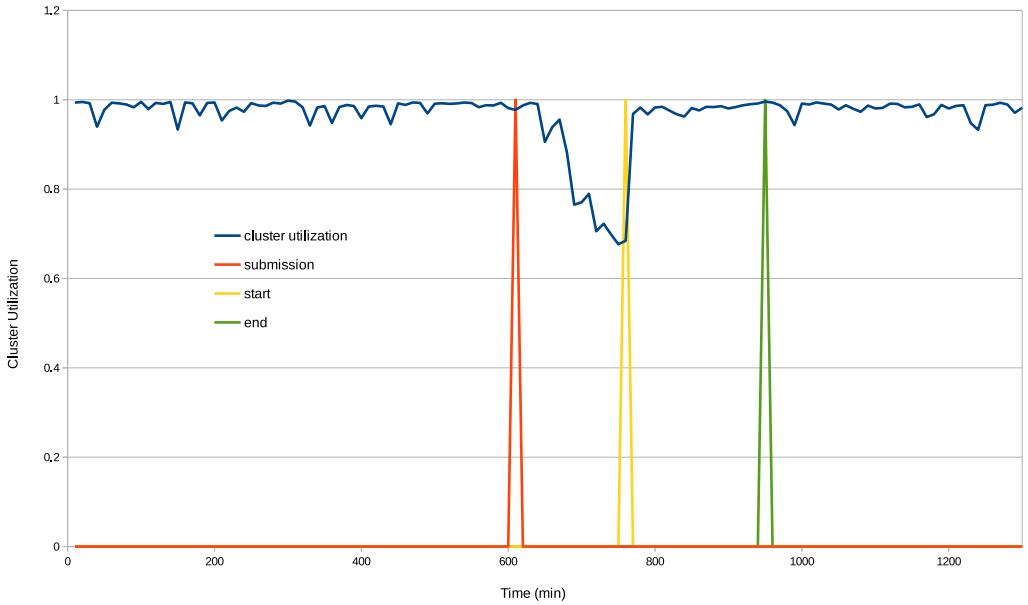

Figure 5.1: Cluster utilization affected by the submission of 31 big jobs. Only the submission (red line) of a high priority 20,000-CPU job for Backfill is shown. The drop in the utilization after this submission is related to the time necessary to reserve enough CPUs to start (yellow line) this job.

Table 5.1: Backfill and Suspend/Resume comparison using Linear plugin. The standard deviation for each metric (5 runs) is shown in parentheses.

Algorithm	WT (s)	RT (s)	SD	WS	UT (%)
Backfill	718.2 (93.3)	123.8 (12.2)	291.7 (51.2)	135.2 (16.0)	97.7 (0.4)
Suspend/Resume	567.0 (85.1)	119.9 (11.8)	227.2 (41.3)	105.1 (11.6)	99.3 (0.4)

more than necessary jobs when a higher priority job is submitted.

We then reconfigured Slurm to operate with the Linear Plugin and we modified the workload combining all jobs that requested less than 16 CPUs into jobs asking for at least this number, and we run again the same experiments. As shown in Figure 5.4, Backfill still presented the same overall behaviour when the 20,000 CPU job was submitted.

However, Figure 5.5 shows a much more improved performance for Suspend/Resume, where only the necessary number of jobs were suspended in order to run a bigger-higher priority job.

The classical metrics were evaluated for five workloads randomly shuffled and the results are shown in Table 5.1. We concluded, based on this data, that Suspend/Resume can be considered a superior scheduling algorithm than Backfill when the workload contains jobs whose size are significant in comparison to the cluster's size.

It is worth noting that Slurm is used in 5 of top 10 HPC systems [Rodrigo et al.,

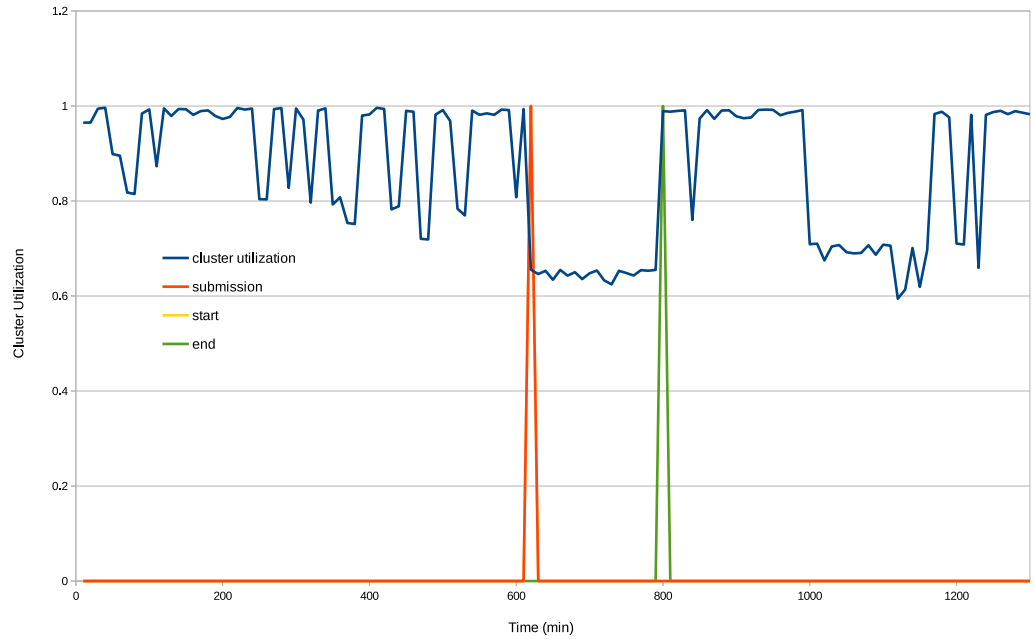

Figure 5.2: Cluster utilization affected by the submission high priority jobs for Suspend/Resume scheduling algorithm. The drops in the utilization coincides with the submission of big-high priority jobs. For comparison with Backfill the submission, the start and end of the 20,000-CPU job is also shown. The start instance (yellow line) coincides with the start (red line) since the Suspend/Resume algorithm starts a high priority job as soon as it is submitted.

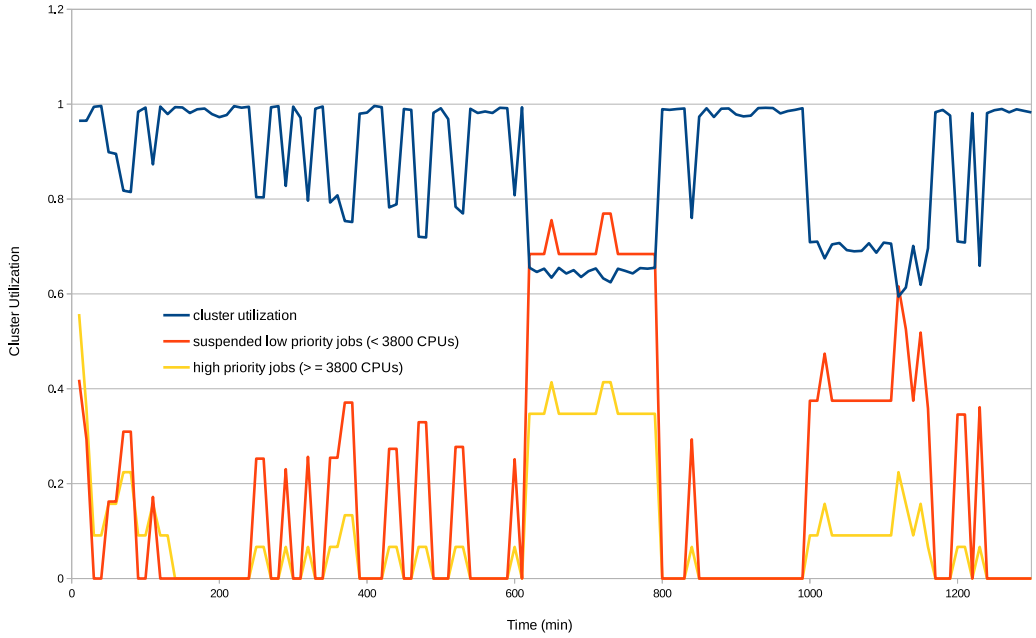

Figure 5.3: Cluster utilization, high priority jobs and suspended jobs. This graph shows that the implementation of Slurm for Suspend/Resume is suspending more low priority jobs than necessary to run a higher priority job.

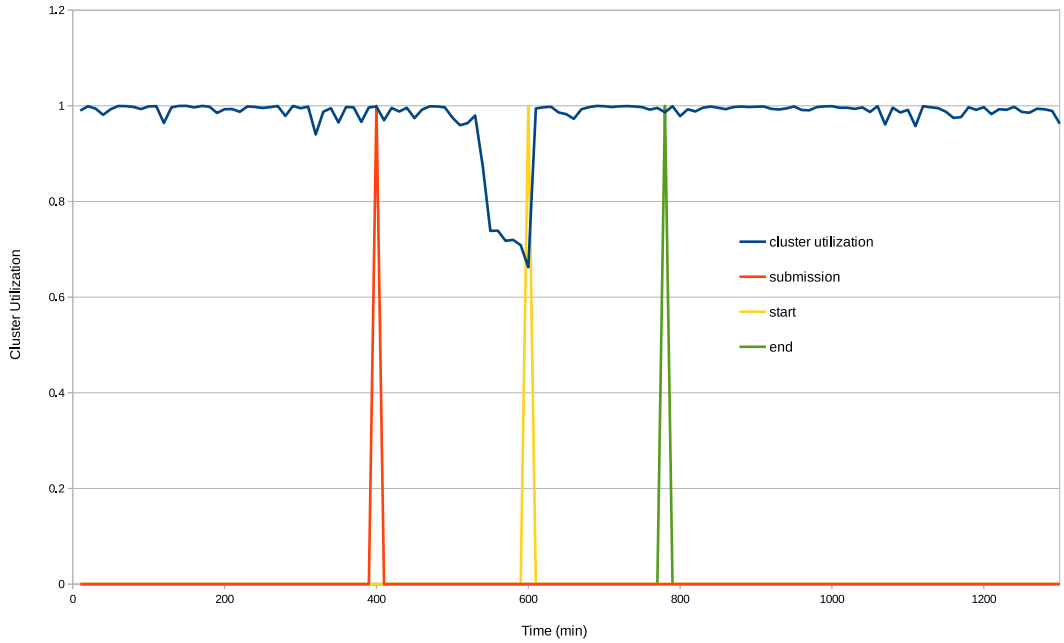

Figure 5.4: Cluster utilization for Backfill for a new workload a utilising the Linear Plugin. All jobs in this workload ask now for at least 16 CPUs (A whole node). The overall behaviour is still the same as presented in Figure 5.1

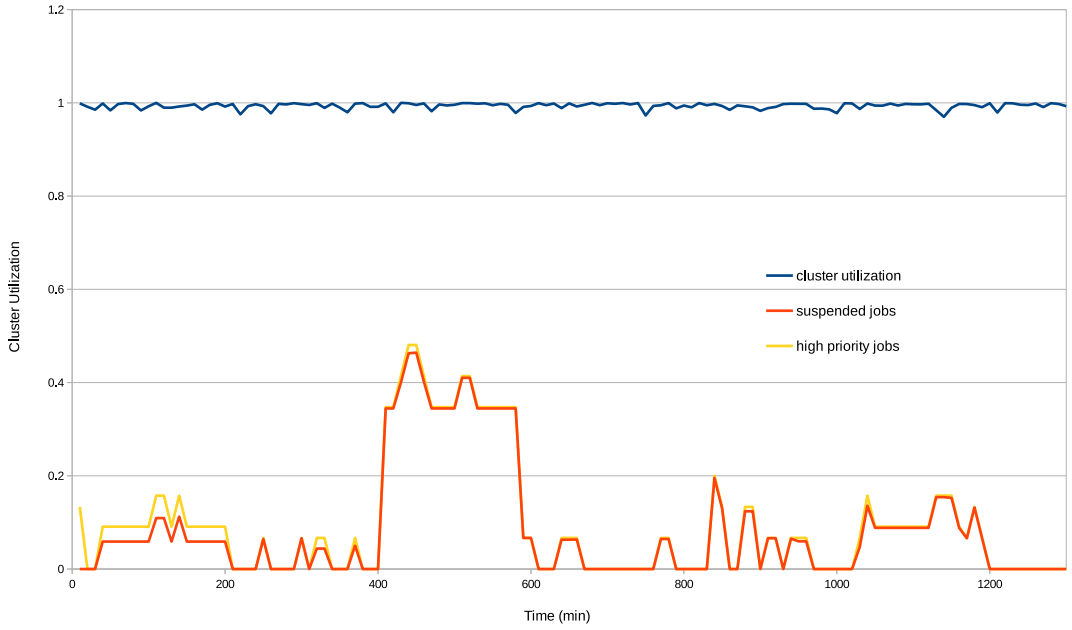

Figure 5.5: Cluster utilization, high priority jobs and suspended jobs. This graph shows the Linear plugin suspends only the necessary number of lower priority jobs to run higher priority jobs, which results in a high cluster utilization.

2017], however this obvious lack of optimization in the implementation of the Consumable Resources plugin, if was ever known , had not been corrected in the Slurm version used in this thesis. This absence of optimization in this plugin would prevent, for instance, the adoption of Slurm at facilities that intend to use the Suspend/Resume technique at the same time as the Consumable Resources plugin.

5.3 Summary

This chapter showed the application of the methodology proposed in the last chapter to evaluate two scheduling algorithms implemented in Slurm. We found that Suspend/Resume showed an overall better performance when compared to Backfill but only for the Linear plugin and only when big jobs are present in the workload. When Slurm was configured to operate with the Consumable Resources plugin, Suspend/Resume suspended more than necessary lower priority jobs.

Chapter 6

Conclusion

Considering that supercomputers are continuously increasing in size, utilization and complexity and the lack of suitable frameworks to support the software development life cycle [Rodrigo et al., 2017], a simple and effective technique to emulate a real size cluster is certainly a valuable tool.

In this thesis, we presented an overview of the past and current scheduling research and introduced an effective framework to support the evaluation of scheduling algorithms. We overcame the limitations of scheduling simulations, like scheduling processing time, communication latencies and implementation complexities by introducing a small set of programs, scripts, and methodologies that emulates a real size computing cluster.

We have shown how a computing cluster hosted on the Nectar cloud with one head node and ten processing nodes running Slurm was able to emulate a 3600 node/57600 CPUs supercomputer, which was very close to the NCI's supercomputer specification.

We also developed a small dummy program, *sigsleep* to play the role of a job during the emulation of a supercomputer's workload. It has a small memory footprint and demands little CPU time. Yet, unlike *sleep*, *sigsleep* keeps track of the suspension periods by detecting and dynamically storing the instants that the suspension and resume signals are sent by Slurm. *Sigsleep* also logs all data related to its run which allows the evaluation of the classical metrics as well the production of all graphs here presented. The job submission was controlled by a script, *run_workload* that read a workload and submitted all *sigsleep* jobs to the cluster. The resulting trace was processed by another program, *calcmets*, that read the trace, evaluated the classical metrics and, optionally, outputted time series data that can be used to produce graphs like the ones shown in this thesis.

We studied four artefacts related to classical metrics; concurrency, submission order, sampling and time shrinking.

Concurrency arises from the non-deterministic nature of time dependent emulations in distributed systems. We found that concurrency indeed affects the values of the classical metrics, but in a small enough degree to conclude that our experiments, though carried out in a distributed system, are reasonably reproducible.

As noticed by different authors, the relative performance of scheduling algorithms depends on the workload used. We have shown that, for Backfill and Suspend/Re-

sume, it is enough to just randomly change the submission order of a single workload to alter the relative performance of their algorithms. In the absence of a strong reason to keep the submission order of a workload unaltered, we suggest that evaluation of the scheduling algorithms be performed using a set of workloads generated by randomly changing the submission order and averaging the classical metrics.

Sampling is a technique to extract a smaller set of jobs from an original workload. Sampling is necessary when the original workload spans a long period of time. Ordering the original workload by respectively expected runtime, runtime and size, and then sampling guarantees the same size distribution. The submission order is automatically guaranteed if the sort algorithm is stable. When the sampling was done in this way the variation of the classical metrics values was found to be smaller than the variations induced by changes in the submission order.

Time shrinking is another technique to further speed up the emulations. We explored in this thesis the limits of time shrinking for the expected and actual runtimes. We discovered that, if the submission time was controlled by the submission script and constant drifts were added to each submission period, we were able to shrink the times up to 90-93%. This resulted in speed ups between 10 to 14 times which are compatible with the results found by [Rodrigo et al., 2017] and [Lucero, 2011].

We used the above methodology to evaluate the implementation of two popular scheduling algorithms in Slurm, Backfill and Suspend/Resume. We found that Suspend/Resume is only superior to Backfill when processing workloads that contain jobs whose sizes are comparable with the cluster's size. In this case, the suspension of smaller and lower priority jobs and the immediate starting of a bigger and higher priority job produces a more optimised utilisation of the cluster and, consequently, better values for the classical metrics. Backfill, on the other hand, needs to keep idle CPUs until it has collected enough CPUs to run the big job, which results in a drastic drop in the cluster utilisation during this period.

Despite the widespread Slurm utilisation in HPC facilities around the world, we found a lack of optimization in its Suspend/Resume implementation. When we configured our cluster to schedule individual CPUs using the Consumable Resources plugin, we noticed that Slurm suspended almost twice the CPUs needed to run a higher priority job. This, obviously, negatively influences the cluster utilisation and classical metrics. On the other hand, the Linear plugin, which selects only whole nodes, had a much better performance and suspended only the necessary number of CPUs in order to run a higher priority job.

6.1 Future Work

One potential direction for the research here presented would be the use of Linux containers to simulate the computing nodes of a supercomputer. Linux containers are a lightweight virtualization technology that uses the Linux kernel and *cgroups* to run multiple Linux systems within a single computer host. A program running inside a container is isolated from all other programs in other containers and it can

be accessed through an internal virtual network.

In this thesis, we used specific Slurm features to emulate a bigger cluster, but the techniques here presented can be easily adapted to be used in a cluster made out of Linux containers. An advantage of such design is the possibility to run the same experiments but using different RMS's like Moab/Torque [Ada, 2017], LoadLever [IBM, 2017] and PBS-pro [Alt, 2017]. Since such experiments would run in the same hardware and software platform they would provide a direct comparison on the performances of different RMS's, as well the quality of their scheduling algorithms implementation.

Appendix A

Scripts

The first script shows the sequence of commands to independently measure the duration of a *sleep 10* command. The *date* command(3) returns epoch time. The output *"Sleeping time: 10"*, as expected, coincides with the time set for *sleep*.

```
1  #!/bin/bash
2
3  # Store the start time in seconds
4  START=$(date +%s)
5
6  # Sleeps 10 seconds
7  sleep 10
8
9  FINISH=$(date +%s)
10 echo "Sleeping time: "$((FINISH-START))
```

(a) testSleep1.

```
1  Sleeping time: 10
```

(b) testSleep1 output.

Figure A.1: Sleep program demonstration using a *bash* script.

The script shown in A.2 mimics the sequence of bash commands starting with *sleep 10*(12) followed by *CRTL-Z* (22), which suspends and sends the program *sleeps* to the background, a waiting period of 5 seconds(25), ending by *fg*(28,31), which resumes the program and bring it to the foreground. If *sleep* was aware of the time that it was suspended the total sleeping time would be 15 seconds but, as the output shows, the total time of this script still continues to be 10 seconds.

```
#!/bin/bash

# Tests the

# Turn on monitor mode (otherwise fg %1 won't work)
set -o monitor

# Store the start time in seconds
START=$(date +%s)

# Sleeps 10 and send it to the background
sleep 10 &

# Store the sleep PID
SLEEP_PID=$!

# A necessary dummy instruction. Otherwise the next kill -TSTP
# command will suspend "sleep 10" before it has time to start.
sleep 0

# Suspend "sleep 10". Same as CRTL-Z
kill -TSTP $SLEEP_PID

# Sleep more 5 seconds
sleep 5

# Resume "sleep 10"
kill -CONT $SLEEP_PID

# Brings the "sleep 10" to the foreground
fg %1

FINISH=$(date +%s)

# If "sleep 10" keeps track of the suspension time
# the whole time would be 10 + 5 = 15 seconds
echo "Sleeping time: "$((FINISH-START))
```

(a) testSleep2.

```
sleep 10
Sleeping time: 10
```

(b) testSleep2 output.

Figure A.2: Demonstration that the *sleep* program is not affect by suspensions.

Appendix B

Slurm Configuration File

Figure B.1 displays a set of selected parameters from the Slurm configuration file which are relevant to the experiments discussed in this thesis.

```
1  ControlMachine=cluster
2  AuthType=auth/munge
3  CryptoType=crypto/munge
4  MaxJobCount=50000
5  ReturnToService=1
6  SlurmctldPidFile=/var/run/slurmctld.pid
7  SlurmctldPort=6817-6836
8  SlurmdPidFile=/var/run/slurmd.%n.pid
9  SlurmdSpoolDir=/var/spool/slurmd.%n
10 SlurmUser=slurm
11 StateSaveLocation=/var/spool/slurmd
12 KillWait=600
13 MessageTimeout=20
14 MinJobAge=10
15 OverTimeLimit=1
16 SlurmctldTimeout=120
17 SlurmdTimeout=300
18 Waittime=0
19 FastSchedule=2
20 SchedulerType=sched/backfill
21 SchedulerPort=7321
22 SchedulerParameters=defer,bf_continue,batch_sched_delay=10,bf_resolution=120,bf_interval=30,sched_min_interval
       =2000000,max_rpc_cnt=100,max_sched_time=5
23 SelectType=select/cons_res
24 SelectTypeParameters=CR_CPU
25 PreemptMode=SUSPEND,GANG
26 PreemptType=preempt/partition_prio
27 ClusterName=cluster
28 SlurmSchedLogFile=/var/slurm/log/slurmsched.log
29 # COMPUTE NODES
30 NodeName=DEFAULT CPUs=16 State=UNKNOWN
31 NodeName=fnode[1-10] NodeHostname=node[1-10] Port=11700
32 ...
33 NodeName=fnode[3591-3600] NodeHostname=node[1-10] Port=12059
34 PartitionName=DEFAULT Nodes=fnode[1-3600] MaxTime=INFINITE STATE=UP Shared=FORCE:1 PreemptMode=suspend
35 PartitionName=low Default=YES Priority=10
36 PartitionName=hig Default=NO Priority=30
```

Figure B.1: Slurm configuration file for backfill experiments.

Appendix C

sigsleep

```
1  // To compile a program that uses clock_gettime, you need to link with librt.a
2  // (the real-time library) by specifying -lrt on your compile line.
3  #include <fcntl.h>
4  #include <inttypes.h>
5  #include <signal.h>
6  #include <stdbool.h>
7  #include <stdio.h>
8  #include <stdint.h>
9  #include <stdlib.h>
10 #include <sys/types.h>
11 #include <string.h>
12 #include <unistd.h>
13 #include <time.h>
14 #include "job.h"
15 #define dbl_ts(x) (x.tv_sec + ( x.tv_nsec / 1000000000.0 ))
16
17 struct job_data job;
18 struct timespec submission, start, end, runtime, suspend, resume, remain,
19                 susp_i, end_trace;
20 struct suspend_period *sus_res = NULL;
21
22 void sighandler(int signum) {
23     sus_res = (struct suspend_period*)malloc(sizeof(struct suspend_period));
24     clock_gettime(CLOCK_REALTIME, &suspend);
25     sus_res->suspend = dbl_ts(suspend);
26     signal(signum, SIG_DFL); // Re-setting the default behavior
27     kill(getpid(), signum);  // Re-sending the signal to obtain the
28                              // default behaviour
29 }
30
31 int main(int argc, char *argv[]) {
32     if (argc != 9) {
33         printf("Wrong␣number␣of␣arguments\n");
34     }
35     job.id = atoi(argv[1]);
36     strcpy(job.prio,argv[2]);
37     job.n_cores = atoi(argv[3]);
38     submission.tv_sec = atoi(argv[4]);
39     submission.tv_nsec = atoi(argv[5]);
```

```
40      remain.tv_sec = atoi(argv[6]);
41      remain.tv_nsec = atoi(argv[7]);
42      end_trace.tv_sec = atoi(argv[8]);
43      end_trace.tv_nsec = 0;
44      job.submission = dbl_ts(submission);
45      job.runtime = dbl_ts(remain);
46      job.orig_runtime = job.runtime;
47      job.sus_res = NULL;
48      job.n_suspensions = -1;
49
50      clock_gettime(CLOCK_REALTIME, &start);
51      job.start = dbl_ts(start);
52      resume = start;
53
54      if (start.tv_sec < end_trace.tv_sec) {
55          do {
56              signal(SIGTSTP, sighandler);
57              job.n_suspensions++;
58              if (job.n_suspensions > 0) {
59                  // job was  suspended and it was now resume. The memory
60                  // allocation and the suspension time were dealt inside
61                  // the singhandler function
62                  clock_gettime(CLOCK_REALTIME, &resume);
63                  sus_res->resume = dbl_ts(resume);
64                  job.total_suspension += sus_res->resume - sus_res->suspend;
65                  sus_res->next = job.sus_res;
66                  job.sus_res = sus_res;
67              }
68              if ((resume.tv_sec + remain.tv_sec) > end_trace.tv_sec) {
69                  // this job would finish after the end of the trace
70                  // Correcting its remaining time and runtime.
71                  if (resume.tv_sec > end_trace.tv_sec) {
72                      // Discount the remainning sleeping time and finish
73                      // the job immediately
74                      job.runtime -= dbl_ts(remain);
75                      break;
76                  } else {
77                      // job corrects its runtime and finishes
78                      job.runtime = job.runtime - (remain.tv_sec -
79                                    (end_trace.tv_sec - resume.tv_sec));
80                      remain.tv_sec = end_trace.tv_sec - resume.tv_sec;
81                  }
82              }
83              runtime = remain;
84          } while (nanosleep(&runtime, &remain) == -1);
85          clock_gettime(CLOCK_REALTIME, &end);
86          job.end = dbl_ts(end);
87      } else {
88          // job started after the end of the trace. In this case lets finish it
89          job.start = dbl_ts(end_trace);
90          job.end = job.start;
91          job.n_suspensions = 0;
92          job.runtime = 0.;
93      }
```

```
94
95      int fd = open("/short/jobs/logs/busySleepLog", O_WRONLY | O_APPEND);
96      struct flock lock;
97      memset(&lock, 0, sizeof(lock));
98      lock.l_type = F_WRLCK;
99      fcntl(fd, F_SETLKW, &lock);
100     write(fd, &job, sizeof(struct job_data));
101     while (sus_res != NULL) {
102         write(fd, sus_res, 2*sizeof(double)); // not writing pointers
103         sus_res = sus_res->next;
104     }
105     lock.l_type = F_UNLCK;
106     fcntl(fd, F_SETLKW, &lock);
107     close(fd);
108     return 0;
109 }
```

Bibliography

2017. Adaptive Computing: moab/torque description. http://www.adaptivecomputing.com. Accessed: September 2017. (cited on page 55)

2017. Altair: pbs-pro description. http://http://www.pbsworks.com/PBSProduct.aspx?n=PBS-Professionali&c=Overview-and-Capabilities. Accessed: September 2017. (cited on pages 6 and 55)

2017. IBM: loadleveler description. https://www-03.ibm.com/systems/power/software/loadleveler/. Accessed: September 2017. (cited on pages 6 and 55)

Atif, M., 2016. Nci cluster documentation. Technical report, National Computational Infrastructure. (cited on page 16)

Atif, M.; Kobayashi, R.; Menadue, B. J.; Lin, C. Y.; Sanderson, M.; and Williams, A., 2016. Breaking hpc barriers with the 56gbe cloud. *Procedia Computer Science*, 93 (2016), 3–11. (cited on pages 5 and 18)

Baker, M. and Buyya, R., 1999. Cluster computing: The commodity supercomputer. *Softw. Pract. Exper.*, 29 (1999), 551–576. (cited on page 5)

Burkimsher, A.; Bate, I.; and Indrusiak, L. S., 2013. Scheduling hpc workflows for responsiveness and fairness with networking delays and inaccurate estimates of execution times. In *European Conference on Parallel Processing*, 126–137. Springer. doi:10.1007/978-3-642-40047-6_15. (cited on page 12)

Conway, R. W.; Maxwell, W. L.; and Miller, L. W., 1967. *Theory of scheduling*. Addison-Wesley. (cited on page 9)

Dunlap, C., 2004. Munge uid n grid emporium. Technical report, Lawrence Livermore National Laboratory. (cited on page 16)

Feitelson, D. G., 1997. Job scheduling in multiprogrammed parallel systems. *IBM Research Report RC 19790 (87657)*, (1997), 1–171. (cited on pages 1, 6, and 7)

Feitelson, D. G. and Rudolph, L., 1995. Parallel job scheduling: Issues and approaches. In JSSPP' 1995: *1st Workshop on Job Scheduling Strategies for Parallel Processing* (Santa Barbara, California, USA, Apr. 1995), 1–18. Springer-Verlag, Berlin, Germany. doi:10.1007/3-540-60153-8_20. (cited on pages 1 and 6)

Feitelson, D. G.; Rudolph, L.; and Schwiegelshohn, U., 2005. Parallel job scheduling: A status report. In JSSPP' 2005: *11th Workshop on Job Scheduling Strategies for*

Parallel Processing (Cambridge, MA, USA, Jun. 2005), 1–16. Springer-Verlag, Berlin, Germany. doi:10.1007/11407522_1. (cited on pages 1, 6, and 11)

Feitelson, D. G.; Rudolph, L.; Schwiegelshohn, U.; Sevcik, K. C.; and Wong, P., 1997. Theory and practice in parallel job scheduling. In JSSPP' 1997: *3rd Workshop on Job Scheduling Strategies for Parallel Processing* (Geneva, Switzerland, Apr. 1997), 1–34. Springer-Verlag, Berlin, Germany. doi:10.1007/3-540-63574-2_14. (cited on pages 1, 6, and 7)

Frachtenberg, E. and Feitelson, D. G., 2005. Pitfalls in parallel job scheduling evaluation. In JSSPP' 2005: *11th Workshop on Job Scheduling Strategies for Parallel Processing* (Cambridge, MA, USA, Jun. 2005), 257–282. Springer, Springer-Verlag, Berlin, Germany. doi:10.1007/11605300_13. (cited on page 29)

Jette, M. A.; Yoo, A. B.; and Grondona, M., 2003. Slurm: Simple linux utility for resource management. In JSSPP' 2003: *9th Workshop on Job Scheduling Strategies for Parallel Processing* (Seattle, WA, USA, Jun. 2003), 44–60. Springer-Verlag, Berlin, Germany. doi:10.1007/10968987_3. (cited on pages 6 and 16)

Klusácek, D. and Rudová, H., 2012. Performance and fairness for users in parallel job scheduling. In *16th Workshop on Job Scheduling Strategies for Parallel Processing* (Shanghai, China, May 2012), 235–252. Springer-Verlag, Berlin, Germany. doi:10.1007/978-3-642-35867-8. (cited on page 12)

Lifka, D., 1995. The anl/ibm sp scheduling system. In JSSPP' 1995: *1st Workshop on Job Scheduling Strategies for Parallel Processing* (Santa Barbara, California, USA, Apr. 1995), 295–303. Springer-Verlag, Berlin, Germany. doi:10.1007/3-540-60153-8_35. (cited on pages 1 and 7)

Lucero, A., 2011. Simulation of batch scheduling using real production-ready software tools. *Proceedings of the 5th IBERGRID*, (2011). (cited on pages 12, 13, 47, and 54)

McGregor, I., 2002. The relationship between simulation and emulation. In *Simulation Conference, 2002. Proceedings of the Winter*, vol. 2, 1683–1688. IEEE. doi:10.1109/WSC.2002.1166451. (cited on page 10)

Mu'alem, A. W. and Feitelson, D. G., 2001. Utilization, predictability, workloads, and user runtime estimates in scheduling the ibm sp2 with backfilling. *IEEE Transactions on Parallel and Distributed Systems*, 12, 6 (2001), 529–543. doi:10.1109/71.932708. (cited on pages 11 and 29)

Niemi, T. and Hameri, A.-P., 2012. Memory-based scheduling of scientific computing clusters. *The Journal of Supercomputing*, 61, 3 (2012), 520–544. doi:10.1007/s11227-011-0612-6. (cited on pages 2, 13, 17, and 36)

Niu, S.; Zhai, J.; Ma, X.; Liu, M.; Zhai, Y.; Chen, W.; and Zheng, W., 2012. Employing checkpoint to improve job scheduling in large-scale systems. In *16th Workshop on*

Job Scheduling Strategies for Parallel Processing (Shanghai, China, May 2012), 36–55. Springer-Verlag, Berlin, Germany. doi:10.1007/978-3-642-35867-8_3. (cited on pages 2 and 11)

Rodrigo, G.; Elmroth, E.; Ostber, P.-O.; and Ramakrishnan, L., 2017. Scsf: A scheduling simulation framework. In *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer. (cited on pages 14, 47, 49, 53, and 54)

Schlagkamp, S., 2015. Influence of dynamic think times on parallel job scheduler performances in generative simulations. In *Proc. of the 19th Intl. Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP'15), part of the 29th IEEE/ACM Intl. Parallel and Distributed Processing Symposium (IPDPS 2015)* (Hyderabad, India, 2015). IEEE Computer Society. (cited on page 12)

Schwiegelshohn, U., 2014. How to design a job scheduling algorithm. In *18th Workshop on Job Scheduling Strategies for Parallel Processing* (Phoenix, Arizona, USA, May 2014), 147–167. Springer-Verlag, Berlin, Germany. doi:10.1007/978-3-319-15789-4_9. (cited on page 7)

Schwiegelshohn, U. and Yahyapour, R., 1998. Analysis of first-come-first-serve parallel job scheduling. In SODA' 1998: *9th annual ACM-SIAM symposium on discrete algorithms*, vol. 98, 629–638. Citeseer. (cited on pages 1, 2, and 6)

Snell, Q. O.; Clement, M. J.; and Jackson, D. B., 2002. Preemption based backfill. In *Workshop on Job Scheduling Strategies for Parallel Processing*, 24–37. Springer. (cited on page 11)

Soner, S. and Ozturan, C., 2013. An auction based slurm scheduler for heterogeneous supercomputers and its comparative performance study. Technical report, PRACE. (cited on page 13)

Soner, S. and Ozturan, C., 2015. Topologically aware job scheduling for slurm. Technical report, PRACE. (cited on page 14)

Tsafrir, D. and Feitelson, D. G., 2006. The dynamics of backfilling: solving the mystery of why increased inaccuracy may help. In *Workload Characterization, 2006 IEEE International Symposium on*, 131–141. IEEE. doi:10.1109/IISWC.2006.302737. (cited on pages 11 and 13)

Yonghong, Y. and Chapman, B., 2008. Comparative study of distributed resource management systems-sge, lsf, pbs pro, and loadleveler. Technical report, Citeseerx. (cited on page 6)

Yuan, Y.; Wu, Y.; Zheng, W.; and Li, K., 2014. Guarantee strict fairness and utilize prediction better in parallel job scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 25, 4 (2014), 971–981. doi:10.1109/TPDS.2013.88. (cited on page 2)