

Predictive Models for Computing Clusters: A Case Study Based on Ilifu



A thesis submitted in partial fulfilment of the requirements for the degree
of

MSc Data Science

By: Jacobus Marthinus Cloete

Supervisor: Dr Bradley Frank

Co-supervisor: Dr Rosephine Rakotonirainy

February 2025

The copyright of this thesis vests in the author. No quotation from it or information derived from it is to be published without full acknowledgement of the source. The thesis is to be used for private study or non-commercial research purposes only.

Published by the University of Cape Town (UCT) in terms of the non-exclusive license granted to UCT by the author.

Plagiarism Declaration

I, Jacobus Marthinus Cloete, hereby declare that the work on which this thesis is based is my original work (except where acknowledgements indicate otherwise) and that neither the whole work nor any part of it has been, is being, or is to be submitted for another degree in this or any other university. I authorise the University to reproduce for the purpose of research either the whole or any portion of the contents in any manner whatsoever.

Acknowledgement

I would like to acknowledge Dr Bradley Frank and Dr Rosephine Rakotonirainy for their constant support and advice, and for assisting me directly with this research.

I would like to acknowledge the use of the ilifu cloud computing facility - www.ilifu.ac.za, a partnership between the University of Cape Town, the University of the Western Cape, Stellenbosch University, Sol Plaatje University, the Cape Peninsula University of Technology and the South African Radio Astronomy Observatory. The ilifu facility is supported by contributions from the Inter-University Institute for Data Intensive Astronomy (IDIA - a partnership between the University of Cape Town, the University of Pretoria and the University of the Western Cape), the Computational Biology division at UCT and the Data Intensive Research Initiative of South Africa (DIRISA).

I would also like to acknowledge Jeremy Smith, Dane Kennedy and Dr Jordan Collier for invaluable support and advice given, for allowing me to attend weekly cluster meetings and for assistance in creating a separate Slurm Training cluster on ilifu to perform my simulations.

Predictive Models for Computing Clusters: a Case Study Based on Ilifu

Jacobus Marthinus Cloete
MSc Data Science
University of Cape Town
February 2025

Abstract

This thesis investigates the feasibility of simulating job waiting times on a High-Performance Computing (HPC) cluster using a simplified discrete event simulation approach. We develop SimPE, a model that uses a Processor Equivalent (PE) metric to reduce the complexity of multi-resource job scheduling to a single dimension. The model is tested against a real Slurm Training Cluster and a Slurm Simulator for various scheduling policies including First-In-First-Out (FIFO), static and dynamic priority, and backfill scheduling. Results show that SimPE effectively models FIFO scheduling with waiting time estimates within 12% of the actual values. For static priority scheduling and dynamic priority scheduling, the model maintains similar levels of accuracy. However, for backfill scheduling, SimPE had more aggressive backfilling which led to much lower waiting time estimates. When modelling jobs with multiple resource requirements, SimPE accurately estimates when one resource dominates, but becomes too conservative when different resources are equally important. This research shows that while PE-based simulation can effectively model certain aspects of cluster behaviour, careful thought is needed when applying it to more complex scheduling scenarios. The findings help to increase our understanding of simple cluster simulation approaches and their practical limitations.

Table of Contents

1	Introduction	1
1.1	Aims and Objectives of the Thesis	2
1.2	Thesis Organisation	3
2	Background	5
2.1	Radio Telescope Background	5
2.1.1	Square Kilometre Array	6
2.1.2	MeerKAT radio telescope	7
2.2	High Performance Computing	10
2.2.1	Computing Cluster	10
2.2.2	Cloud Computing	10
2.2.3	Astronomy Computing Requirements	12
2.2.4	Ilifu Computing Cluster	12
2.3	Slurm Job Scheduler	13
2.3.1	Submitting a Computing Job	14
2.3.2	Partitions	15
2.3.3	Scheduling	16
2.3.4	Fairshare	19

3	Prediction of Queue Wait Time	23
3.1	Queuing Theory	23
3.1.1	M/M/1 Queue	24
3.1.2	M/M/1 with Priority Queuing	25
3.1.3	G/G/1 Queue	26
3.2	Discrete Event Simulation	27
3.2.1	Waiting Time Distributions	27
3.2.2	Processor Equivalent Function	30
3.3	Related Work	32
4	Methodology	34
4.1	SimPE Simulation Model	35
4.2	Software	36
4.2.1	Simpy	37
4.2.2	Salabim	38
4.3	Key Model Iterations	39
4.3.1	Simpy: Proof of concept model	40
4.3.2	Salabim: FIFO and Static Priority Queuing	41
4.3.3	Salabim: Dynamic Job Priority	41
4.3.4	Salabim: Backfilling	43
4.4	Modelling Detail	44
5	Testing and Implementation	50
5.1	Slurm Training Cluster	51
5.2	Slurm Simulator	53

5.3	Job Batch Assumptions	56
5.3.1	Job Inter-arrival Times	57
5.3.2	Job Run Times	57
5.3.3	System Utilisation Tuning	58
5.4	Test Simulation Runs	59
5.4.1	FIFO	60
5.4.2	Priority	61
5.4.3	Static Priority	61
5.4.4	Dynamic Priority	62
5.4.5	Backfill	63
5.4.6	Multi-resource	66
5.5	Selected Job Configurations	67
6	Results	69
6.1	FIFO	69
6.2	Priority	71
6.2.1	Static Priority	71
6.2.2	Dynamic Priority	74
6.3	Backfill	75
6.4	Multi-Resource	78
6.4.1	Three Job Types	78
6.4.2	Four Job Types	80
7	Discussion and Conclusion	83
7.1	Key Findings	84

7.2	Limitations of the Thesis	85
7.3	Future Studies	88
7.4	Code Availability	91
A	Appendix: SimPE Modelling Detail	97
B	Appendix: Slurm Training Cluster Configuration	99

List of Tables

5.1	Job types used in FIFO simulation runs.	60
5.2	Job types used in Static Priority runs.	62
5.3	Job types used in Backfill runs.	64
5.4	Job types used in Multi-resource runs (three job types version).	67
5.5	Job types used in Multi-resource runs (four job types version).	67
6.1	Comparison of the mean job waiting time (in seconds) for FIFO scheduling between the SimPE model and <i>Slurm Training Cluster</i>	70
6.2	Comparison of the mean job waiting time (in seconds) by job type between the <i>Slurm Training Cluster</i> and SimPE model using FIFO scheduling.	71
6.3	Comparison of the mean job waiting time (in seconds) for Priority scheduling between the SimPE model and the <i>Slurm Training Cluster</i>	72
6.4	Comparison of the mean job waiting time (in seconds) by job type between the <i>Slurm Training Cluster</i> and SimPE model using Priority scheduling.	72
6.5	Comparison of the mean job waiting time (in seconds) by job type between the SimPE model and <i>Slurm Training Cluster</i> using Dynamic Priority scheduling.	74

6.6	Comparison of the mean job waiting time (in seconds) for Backfill scheduling between the SimPE model and <i>Slurm Training Cluster</i>	76
6.7	Comparison of the mean job waiting time (in seconds) by job type between the SimPE model, <i>Slurm Simulator</i> model and the <i>Slurm Training Cluster</i> , all using Backfill scheduling.	76
6.8	Comparison of the mean job waiting time (in seconds) for Multi-resource runs (with three job types), between the SimPE model and the <i>Slurm Training Cluster</i>	79
6.9	Comparison of the mean job waiting time (in seconds) by job type between the <i>Slurm Training Cluster</i> and SimPE model for Multi-resource runs.	79
6.10	Comparison of the mean job waiting time (in seconds) for Multi-resource runs (with four job types), between the SimPE model and the <i>Slurm Training Cluster</i>	80
6.11	Comparison of the mean job waiting time (in seconds) by job type between the <i>Slurm Training Cluster</i> and SimPE model for Multi-resource runs.	81

List of Figures

2.1	Image of the MeerKAT radio telescope array close to Carnarvon, South Africa.	7
2.2	An image of giant bubbles of radiation in the centre of the Milky Way galaxy by the MeerKAT telescope.	8
2.3	A flowchart of the MeerKAT radio telescope's data flow, with data flow depicted by solid black lines.	9
2.4	An image inside the Ilifu computing cluster shows the main server corridor with the computing nodes in racks and cooling system overhead.	13
2.5	An overview of Slurm components.	14
2.6	An illustration of Backfill scheduling.	17
2.7	An illustration of Ilifu's share distribution.	20
2.8	An example of Level Fairshare Calculation.	21
3.1	A simple queueing model for computing jobs.	24
3.2	A Markov queueing model with arrival rate of λ and completion rate of μ	24
3.3	A Markov queueing model (M/M/1) with an arrival rate of $\lambda=0.4$ and completion rate of $\mu=0.5$	28
3.4	The service time and resulting waiting time distribution for a queueing system with exponential service times, based on 10,000 samples.	28

3.5	The waiting time distribution for a uniform service time distribution.	29
3.6	The waiting time distribution for a normal service time distribution.	29
3.7	An illustration of two different computing jobs with a similar impact on a computing node, even though they have different resource requirements.	31
3.8	The ilifu cluster usage per week in 2021.	32
4.1	This plot shows the available PE as a function of time as jobs are submitted.	35
4.2	An illustration of the FIFO job model that was implemented with <i>Simpy</i>	40
4.3	An illustration of the static priority job model that was implemented with <i>Salabim</i>	41
4.4	An illustration of the static priority job model that was implemented with <i>Salabim</i>	42
4.5	An illustration of a backfill job model that was implemented with <i>Salabim</i>	43
4.6	An illustration of the backfill calculation process.	44
5.1	A diagram of the <i>Slurm Training Cluster</i> , with nodes as rectangles and communication protocols between the nodes indicated with solid lines and arrows.	51
5.2	A flow diagram of the Simakov Slurm simulator showing the input files required by the model and the generated output files.	55
6.1	A plot of the job waiting times for the first FIFO simulation run comparing the <i>Slurm Training Cluster</i> and SimPE model.	70
6.2	A plot of the job waiting times in Priority run number 1 for the <i>Slurm Training Cluster</i> and SimPE model.	73

6.3	A plot of the job waiting times in Backfill run number 1 for the <i>Slurm Training Cluster</i> and SimPE model.	77
6.4	A plot of the job waiting times in Backfill run number 1 for the <i>Slurm Training Cluster</i> and <i>Slurm</i> Simulator.	78
6.5	A PE Time Plot of the first 25 minutes of the Multi-resource simulation run with 3 Job types between the <i>Slurm Training Cluster</i> and the SimPE model.	80
6.6	A PE Time Plot of the first 25 minutes of the Multi-resource simulation run with 4 job types between the <i>Slurm Training Cluster</i> and the SimPE model.	82

Dedication

This thesis is dedicated to my wife, Clara, for her constant support through everything. Without her, this thesis would definitely not have been completed. Thank you for not running away and for keeping me motivated. Agteros kom ook in die kraal.

Chapter 1

Introduction

The Square Kilometer Array (SKA) will be the largest telescope in the world, and is designed to address several of the most exciting open questions in cosmology and galaxy evolution (Johnston et al., 2007). Its mid-frequency component, SKA-Mid, located in South Africa, will consist of hundreds of dishes, requiring the processing of massive volumes of data, potentially reaching exabytes (D’Isanto et al., 2018). This data, primarily imaging and interferometric in nature (Onose et al., 2016), requires the use of advanced High Performance Computing (HPC) clusters (Mao et al., 2016). However, the scale of data processing expected for the SKA presents unprecedented challenges for current HPC technologies (Broekema, Nieuwpoort, et al., 2015). To address these issues, precursor telescopes, such as South Africa’s MeerKAT, initially an SKA pathfinder, have been instrumental. MeerKAT, now the world’s most sensitive mid-frequency radio telescope, has facilitated significant scientific advancements and aids in understanding and resolving the substantial computational demands posed by projects like the SKA (Jonas et al., 2016; Jozsa et al., 2022; Rajwade et al., 2022). An example of a computing cluster created in the process of supporting processing for the high volumes of MeerKAT data is the Ilifu¹ computing cluster, which is operated by the Inter-University Institute for Data Intensive Astronomy (IDIA). In addition to supporting astronomy projects, the Ilifu computing cluster also assists bioinformatics projects, showcasing its versatile application across different scientific domains.

This thesis focuses on the Ilifu computing cluster, a multi-disciplinary facility used for data-intensive research, particularly in fields like astronomy

¹<https://idia.ac.za/ilifu-research-cloud-infrastructure/>

and bioinformatics. The Ilifu cluster, employs the *Slurm*² Job Scheduler for the allocation and scheduling of computing resources. Ilifu faces a significant challenge, namely the variability and unpredictability of waiting times for computing jobs, especially during periods of high demand. The issue of job waiting time prediction is a known problem in the broader HPC community, as shown by previous research (Downey, 1997; W. Smith et al., 1999; H. Li et al., 2004; Murali et al., 2016). However, on the Ilifu cluster, this problem is exacerbated due to several factors. Firstly, the cluster experiences high job variability resulting from its diverse user base, which often has very different resource requirements. Secondly, Ilifu employs a scheduling system that prioritises jobs based on multiple factors. This system, known as Slurm’s multi-priority plugin³, incorporates a fair-share factor to ensure equitable distribution of cluster resources among various stakeholders. While this approach aims to optimise resource allocation, it also introduces additional complexity to the scheduling process, making it more challenging to predict job waiting times accurately (Murali et al., 2016). The combination of these factors – high job variability, diverse resource requirements, and a complex, dynamic scheduling system – creates a unique environment where traditional approaches to job waiting time prediction fall short. This situation underscores the need for a more sophisticated model that can capture the intricacies of Ilifu’s specific operational context. The goal of this project is to take a first step in attempting to model the scheduler behaviour on the Ilifu cluster in a way that could help improve understanding of the drivers of waiting time. By developing such a model, we aim to provide insights that could lead to more efficient resource allocation, reduced waiting times, and ultimately, enhanced productivity for the diverse range of scientific projects that rely on the Ilifu cluster.

1.1 Aims and Objectives of the Thesis

The primary aim of this thesis is to develop a simulation model, referred to as SimPE, that mimics the functionality of Ilifu cluster’s Slurm job scheduler. This model is designed to enhance our understanding of job waiting times and potentially improve prediction capabilities. To achieve this aim, the following specific objectives have been identified:

²<https://slurm.schedmd.com/overview.html>

³https://slurm.schedmd.com/priority_multifactor.html

1. To develop the SimPE model using a Discrete Event Simulation (DES) approach, incorporating the novel Processor Equivalent (PE) metric as a simplification of multi-resource allocation.
2. To validate the SimPE model against a real or simulated Slurm system.
3. To assess the usefulness of the modelling Slurm using a PE-based DES model

It is important to note that while the Ilifu cluster’s scheduling system includes a fairshare factor, this aspect has been classified as out of scope for the current study due to time constraints and the complexity of implementation and validation. This decision allows for a more focused analysis within the limitations of a mini-dissertation format, while still aiming to provide valuable insights into the cluster’s behaviour. While it is beyond the scope of this thesis, we provide a brief review of fairshare and its impact on job scheduling in Chapter 2, for the sake of completeness.

1.2 Thesis Organisation

The rest of this thesis is organised in six chapters.

- **Chapter 2: Background.** Provides an overview of radio astronomy projects like SKA and MeerKAT, concepts in High Performance Computing, and more details about the Slurm job scheduler used on the Ilifu cluster.
- **Chapter 3: Prediction of Queue Wait Time.** In this chapter, we review the theoretical foundations of queue waiting time prediction, including queueing theory and discrete event simulation. It also discusses related work in HPC job scheduling.
- **Chapter 4: Methodology.** The development of the SimPE simulation model, including its requirements, implementation details, and key iterations, is described in this chapter.
- **Chapter 5: Testing and Implementation.** Outlines the setup of the *Slurm Training Cluster* and the Slurm Simulator, and details the assumptions and methods used for testing the SimPE model.

- **Chapter 6: Results.** Presents the findings from various testing runs, comparing the SimPE model's performance against the *Slurm Training Cluster* and Slurm Simulator.
- **Chapter 7: Discussion and Conclusion.** Analyses the results, discusses the limitations, draws conclusions about the effectiveness of the SimPE model, and suggests directions for future research.

Chapter 2

Background

In this chapter, we provide an overview of the challenges faced in High Performance Computing (HPC) for radio astronomy, particularly focusing on the Square Kilometre Array (SKA) and the MeerKAT radio telescope. Understanding the context and background of these telescopes is crucial, as they do not just represent milestones in radio astronomy but also come with unique challenges in data handling and computation. We then provide more details about the Ilifu computing cluster, and provide a detailed explanation of the Slurm job scheduler, the software used to manage computational resources on the Ilifu cluster. In particular, we discuss the different scheduling methods in Slurm, and those used by Ilifu. This background is an important context for the discussion of the modelling of job waiting times on the cluster, and specifically the SimPE simulation that is developed in this research project.

2.1 Radio Telescope Background

This section will focus on two major radio astronomy projects: the SKA and the MeerKAT radio telescope. These instruments are the pinnacle of modern radio astronomy and are key to understanding the universe. This section will set the stage for discussing how these challenges are being addressed and how they relate to the broader objectives of this research.

2.1.1 Square Kilometre Array

The Square Kilometre Array (SKA) is a global effort to build the world’s most capable radio telescope, which will eventually have a combined collecting area of about one square kilometre (Bourke et al., 2015; Grainge et al., 2017; Broekema, Van Nieuwpoort, et al., 2012). Current estimates indicate that the SKA will be 50 times more sensitive than any other existing radio telescope (Johnston et al., 2007). The project is funded by the SKA Organisation, which, as of July 2023, was made up of 16 member countries¹.

The SKA will consist of hundreds of radio dishes and hundreds of thousands of small antennas. The small antennas will cover the lower frequencies of the SKA range (SKA-Low), and the radio dishes will cover the middle frequencies (SKA-Mid) (Grainge et al., 2017). The signals received from these different receivers are combined in such a way that they simulate a larger radio telescope, called array synthesis (G. B. Taylor et al., 1999, Chapter 5).

The project will be built in phases, and Phase 1 of SKA will focus on the completion of infrastructure in South Africa and Australia (A. Taylor, 2012). The low-frequency antenna array will be located in Western Australia, where progress has already been made by building two precursor antenna arrays called the Murchison Widefield Array (MWA) and the Australian SKA Pathfinder (ASKAP) (Tingay et al., 2012). Mid-frequency radio dishes are being built in the Karoo Desert of South Africa, where two precursor telescopes have already been constructed in the area, namely MeerKAT and the Hydrogen Epoch of Reionization Array (HERA) (Jonas et al., 2016; DeBoer et al., 2017). These precursor facilities inform or drive the design of data centres, and understanding the scale of the data problem (Tingay et al., 2012).

The second phase will further extend the telescope, with additional sites built in up to 8 African countries². This will increase the total collecting area of the telescope by a factor of 10 (A. Taylor, 2012). A larger collecting area makes the telescope more sensitive and able to observe fainter objects (Fanaroff, 2022).

¹<https://www.skao.int/en/partners>

²https://www.sarao.ac.za/wp-content/uploads/2017/06/presentation_parliament_2017.pdf

The completed SKA will enable scientists to study the emergence of the first stars and black holes. It will also enable more robust testing of the theory of relativity, the study of the evolution of galaxies, and dark energy (A. Taylor, 2012).

The SKA and other projects like the Large Synoptic Survey Telescope (LSST) are simply a few examples that will require exabyte-level data processing and storage (D’Isanto et al., 2018). Even the latest technology in high performance computing are challenged by these requirements (Broekema, Nieuwpoort, et al., 2015).

Traditionally, the astronomy community solved computational problems with custom hardware and software. However, the increase in computational resources available in commercial cloud computing providers also provides an opportunity to help address these challenges (Amanda E. Bauer et al., 2019; A. M. Smith et al., 2019).

2.1.2 MeerKAT radio telescope

MeerKAT is a radio telescope consisting of 64 dishes that is located in the Meerkat National Park, close to Carnarvon, South Africa (Merwe et al., 2021; Jonas et al., 2016). The telescope was originally built as an SKA pathfinder and was launched in 2018 (Jonas et al., 2016). It is currently the most sensitive mid-frequency radio telescope in the world (Jonas et al., 2016).



Figure 2.1: Image of the MeerKAT radio telescope array close to Carnarvon, South Africa. Source: <https://www.sarao.ac.za/gallery/meerkat>.

The telescope will eventually integrate into the SKA 1 mid-frequency array, along with another 197 dishes (A. Taylor, 2012). However, MeerKAT will

operate as an independent telescope until this integration occurs. Since its launch, MeerKAT has already been used for great scientific observations. An example is the image unveiled at MeerKAT’s launch of giant bubbles of radiation in the centre of the Milky Way (Heywood et al., 2022) (see Figure 2.2). It has also performed deep-sky surveys of portions of the sky at fainter levels than before.



Figure 2.2: An image of giant bubbles of radiation in the centre of the Milky way galaxy by the MeerKAT telescope. Source: <https://www.sarao.ac.za/gallery/meerkat>.

The MeerKAT telescope generates very large volumes of data that require fast processing. Figure 2.3 shows a flowchart of MeerKAT’s data flows. The main subsystems that are shown in the flowchart include the MeerKAT antennas, correlator and beamformer (CBF) and Science Data Processing (SDP) subsystem (Jonas et al., 2016). The data collected from all the MeerKAT antennas is sent to the CBF for real-time processing. The CBF performs multiple functions, including splitting the data into multiple frequency bands (called channelisation), correlating and averaging the signals of each pair of antennas and creating virtual beams for observing the sky (called beamforming) (Joubert et al., 2017; Brederode et al., 2016; Jonas et al., 2016). Thereafter, the data is sent for science data processing which includes running an image processing pipeline for calibrating and cleaning the data to create image cubes (Brederode et al., 2016). The running of the processing pipeline does not have to be done in real time, and is therefore done in Cape Town (Brederode et al., 2016). The data is transferred from Carnarvon to the CHPC/SARAO data archive in Cape Town using a long-distance fibre optic cable (Glass, 2020). For most of the MeerKAT science cases, a more optimised approach to calibration and imaging is required. For these use-cases, the data is transferred from the

SARAO Archive to IDIA’s Ilifu computing cluster (discussed further in Section 2.2.4). After running the image calibration pipeline, the output is saved on Ilifu. Thereafter, the data is moved to a science archive and made publicly available (Swart et al., 2022).

Currently, for every one hour of observational data recorded by the MeerKAT radio telescope, about 1 TB³ of data is generated that has to be processed by the imaging pipeline. When the SKA 1 comes fully online, the number of antennas is expected to increase from 64 to 261 (64 + 197) (A. Taylor, 2012). This would increase the number of baselines (i.e., unique number of pairs of antennas) of the radio telescope from 2016 ($\frac{64*63}{2}$) to 33,930 ($\frac{261*260}{2}$), which is almost 20 times larger. The data rate would be expected to increase by a similar ratio, from 1 TB to 20 TB³ per hour. However, even the current levels of MeerKAT data are already challenging to handle, and would therefore require many advances in the hardware and software used. In the next section, we will discuss the High Performance Computing systems that are used to perform the MeerKAT’s image processing and astronomy research.

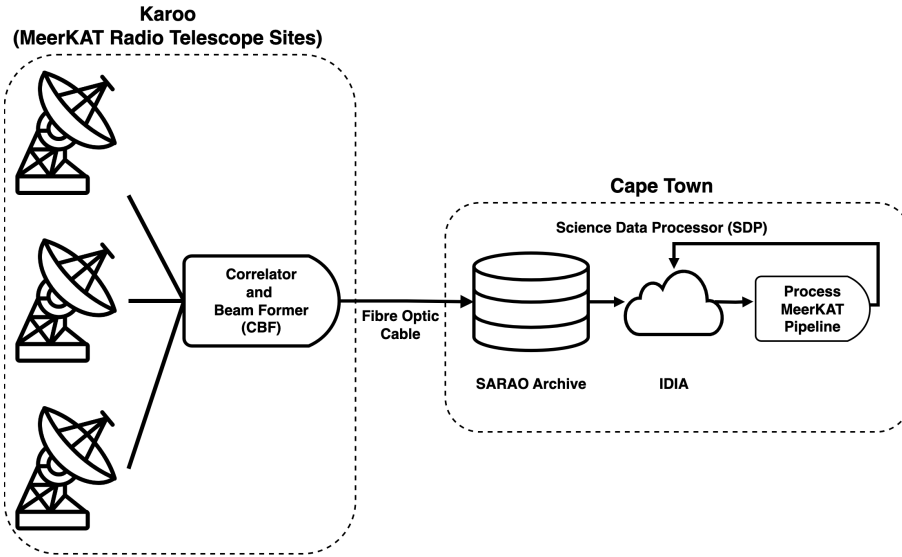


Figure 2.3: A flowchart of the MeerKAT radio telescope’s data flow, with data flow depicted by solid black lines. The raw data is collected at each MeerKAT dish, and transported to the correlator, where it is formed into the basic measurement set. This data is transported via fibre optic cable to the SARAO archive. Most data requires calibration and imaging, for which it is transferred from the SARAO Archive to IDIA’s Ilifu computing cluster. An image calibration pipeline is then run on the data, with the results saved on Ilifu.

³We assumed the MeerKAT radio telescope was used in 32,768 channel mode and recording with 2 polarisations at a visibility integration period of 8 seconds. The total number of samples was then calculated by using $N_{\text{samples}} = N_{\text{baselines}} \times N_{\text{time_samples}} \times N_{\text{polarisations}} \times N_{\text{channels}}$. With $N_{\text{baselines}} = 64 * 63 / 2 = 2016$ and $N_{\text{time_samples}} = 60 * 60 / 8 = 450$. This gives $N_{\text{samples}} = 59,454,259,200$. Assuming that each sample value is stored in 16 bytes (e.g. two 8-byte floating point numbers), this gives a total data volume of close to 1 TB per hour ($N_{\text{samples}} * 16 / 10^{12}$) per hour. A similar estimation performed on 261 antennas yields an estimated data rate of 20 TB per hour

2.2 High Performance Computing

In this section, we give an overview of High Performance Computing (HPC), which is especially important for modern scientific research. This is particularly for fields that generate and process vast amounts of data, like radio astronomy (Grainge et al., 2017; Broekema, Van Nieuwpoort, et al., 2012). In this thesis, HPC refers to various technologies, from traditional computing clusters to contemporary cloud computing solutions (Sterling et al., 2017). We will explore how these technologies are used to handle the enormous computational demands of projects such as the SKA and MeerKAT telescopes.

2.2.1 Computing Cluster

High performance computing (HPC) is the discipline of using computers and computer clusters to perform complex and large scale calculations (Sterling et al., 2017). For radio astronomy, using a cluster of computers enables large amounts of data to be processed in a timely fashion. A computing cluster is a network of computers connected to function as a single entity (Prabhu, 2008). This allows computations to be run that require more memory or processing power than possible on a single computer or server (Baker et al., 1999). Normally, users access the cluster by creating requests for computational resources. A centralised job scheduler then allocates computational resources to user requests (Prabhu, 2008). The nodes (or different computers) in a computing cluster often have the same operating system and similar hardware (Baker et al., 1999). This makes it easier to split the same computing job over multiple nodes.

2.2.2 Cloud Computing

Modern astronomy computing requires a combination of intensive computing and a flexible framework for analysis (Araya et al., 2018). An ideal solution to these different demands is to use cloud computing. Cloud computing involves assembling a large number of diverse and often non-geolocated computing resources under a single operating framework (Islam et al., 2023). In practice, this involves the running of a communication and management service on the constituent nodes, combined to present a con-

solidated computing "cloud". *Openstack*⁴ is a popular collection of open source software that can be used to create a private cloud implementation. This gives the service provider, for example, an academic institution, the flexibility to apportion chunks of computing resources according to their users' requirements, often folding the computing back into the "pool" after completion of the use-case⁵.

Cloud computing became mainstream with the rise of Amazon, Google, and Microsoft in the early 2000s (Brian et al., 2008). Amazon was the first to launch its Amazon Web Service (AWS) cloud computing platform in 2002 and provided public access to storage and computing in 2006 (Qian et al., 2009). Microsoft followed with Microsoft Azure in 2008 (Qian et al., 2009). Google launched Google Cloud Platform in 2008 and added computing and storage access in 2013 (Z. Li et al., 2013).

This widespread availability of cloud computing resources has caused disruptive changes in the way IT services are provided (Sunyaev, 2020). The size and specialisation of these commercial companies allow them to benefit from economies of scale, which allows competitive pricing of resources (Bottum et al., 2017). A major benefit of utilising a cloud provider is the increased scaling flexibility, where more resources can be requested as required. Cloud providers can provide this flexibility due to the size of the resource pool; however, during low demand periods, this leads to idle resources. In order to solve this, different pricing models are typically used. As an example, Amazon AWS has 'reserved', 'on demand', and 'spot prices' for computational (and other) resources (Javadi et al., 2013). The 'reserved price' is the reduced price given for long term commitments (e.g., multiple years). The 'on demand price' is the fixed price given per time unit that the resources are used. The 'spot price' is a significantly reduced price for spare resources in the pool, while there are spare resources available (Portella et al., 2019).

In general, due to the way cloud computing prices the usage of resources, there seem to be certain problems that it is economically more suited to solving. For instance, problems that are embarrassingly parallel in nature, where the problem can easily be split into several parallel tasks, are generally easier and cheaper to run in the cloud (Bottum et al., 2017). Applications running with bursts of high computation, followed by periods

⁴<https://www.openstack.org/>

⁵<https://www.techtarget.com/searchcloudcomputing/definition/OpenStack>

of low computation, also work especially well. However, for longer periods of sustained computation, the cost often has to be mitigated by using spot instances (Byrne et al., 2021). Long term storage of large data sets can also become expensive, especially when the cost of data egress is included (A. M. Smith et al., 2019).

2.2.3 Astronomy Computing Requirements

Due to the burgeoning data rate produced by large scale astronomical facilities, astronomers are further motivated to move towards a flexible model of computing, which can be scaled according to their use case (Johnston et al., 2007; Bottum et al., 2017). Astronomy use-cases include large scale and compute/memory/storage intensive data processing, detailed analyses of astronomical data that is highly labour intensive and is often manually done by the astronomer, and large scale simulations of astrophysical phenomena or telescope performance (Bogert et al., 2014; Khalil et al., 2019). Since these use-cases present diverse requirements to a data centre operator, cloud computing is rapidly becoming the preferred mode of operation due to its flexibility (Bottum et al., 2017; Islam et al., 2023).

2.2.4 Ilifu Computing Cluster

Ilifu⁶ is a cloud computing platform in South Africa. It is managed by IDIA⁷, the Inter-university Institute for Data Intensive Astronomy. Its main focus is to support data-intensive research in both the astronomy and bioinformatics fields⁶. It is based on an open stack cloud computing platform¹. Ilifu provides several specialised computing partitions to meet different research needs: a main compute cluster, high-memory nodes, GPU-enabled nodes, Jupyter notebook servers, and a development partition¹. The infrastructure consists of approximately 110 computing nodes, each with 32 CPUs and 232 GB of memory, with about 94 dedicated to batch processing⁸. Figure 2.4 shows an image of the Ilifu computing nodes.

Users can access these resources through either batch job submission or interactive Jupyter sessions. The analytics and development platform is

⁶<https://www.ilifu.ac.za/about/>

⁷<https://www.idia.ac.za>

⁸https://docs.ilifu.ac.za/#/tech_docs/specifications

primarily provided through hosted Jupyter notebooks¹. Since Ilifu serves multiple funders and stakeholders, a fair sharing policy is implemented to balance the interests of different users¹.

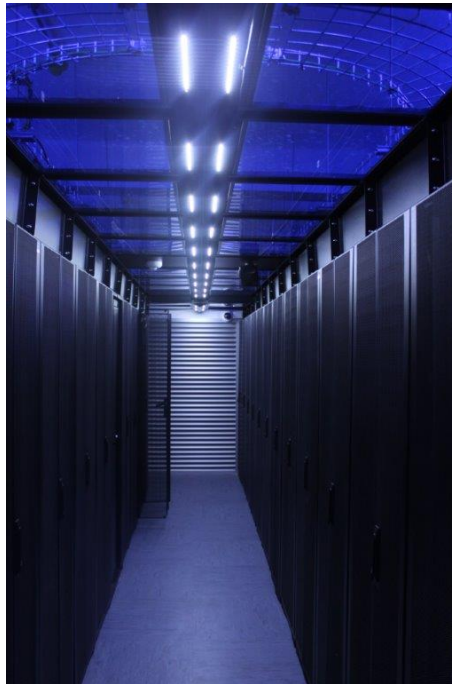


Figure 2.4: An image inside the Ilifu computing cluster shows the main server corridor with the computing nodes in racks and cooling system overhead. Source: <https://uct.ac.za/eresearch/articles/2018-12-07-ilifu-data-intensive-research-cloud-ready-first-bioinformatics-and-astronomy>

One of the primary use cases for the Ilifu infrastructure is to serve as the main destination for processing and analysing data from South Africa’s MeerKAT radio telescope. Ilifu assists in both the MeerKAT Open Time Projects and the Meerkat Large Survey Projects. In fact, five of the eight Meerkat Large Survey projects are served by Ilifu⁹. Each large survey project comprises hundreds of international researchers. In fact, five of the eight Meerkat Large Survey projects are served by Ilifu¹⁰, which suggests that Ilifu services the majority of MeerKAT’s users, in addition to a large community of open time users.

2.3 Slurm Job Scheduler

The Ilifu computing cluster’s many users must be given access to the computing nodes to perform computations. The software¹¹ used by Ilifu for

⁹<https://www.ilifu.ac.za/research/>

¹⁰<https://www.ilifu.ac.za/research/>

¹¹https://docs.ilifu.ac.za/#/getting_started/access_ilifu

this task is the *Slurm Workload Manager*², simply known as *Slurm*. *Slurm* is an open source job scheduling system for Linux computing systems. In general, a job scheduler is software that receives computing jobs and then schedules the order in which the computing jobs will be processed (Aida, 2000). *Slurm* is widely used among the Top 500 supercomputers, with some estimates being higher than 50%¹².

2.3.1 Submitting a Computing Job

The main aim of this project is to predict the waiting time of computing jobs in this pending job queue. Therefore, it is very important to understand how *Slurm* manages its job queues.

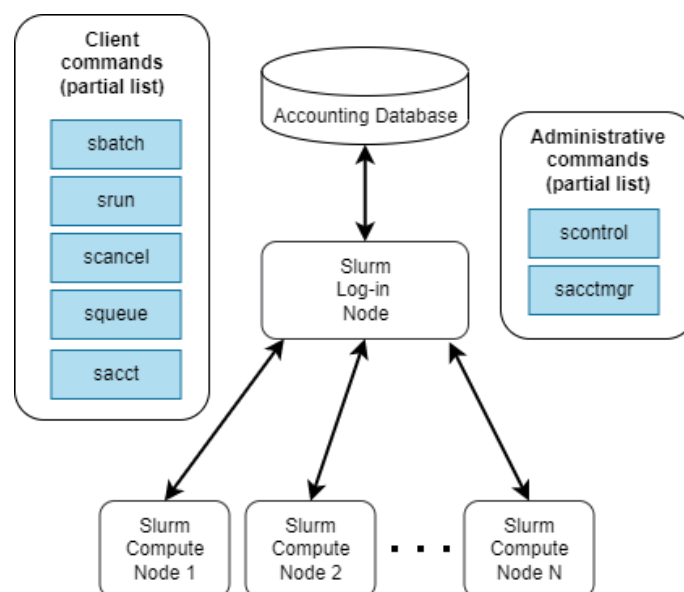


Figure 2.5: An overview of *Slurm* components. In the top left box is a partial list of the most commonly used *Slurm* commands. In the top right is a partial list of the most common *Slurm* administrative commands. The flowchart depicts the typical and simplified communication and management layout used by *Slurm*; users log in to the Log-in node, which often doubles-up as the head or controller node. Cluster resources are tracked and logged by the accounting database, computing commands or applications or jobs are executed on the allocated compute nodes via the *Slurm* scheduler.

On a user level, *Slurm* operates as a collection of utilities that a user can use to manage their computing jobs. Figure 2.5 gives an overview of the different *Slurm* components and user utilities. In order to access the system, users log in to a *Slurm* log-in node¹¹. Resource requests are then submitted using the *Slurm* utilities². The most often used *Slurm* utilities enable the user to submit batch computing jobs (*sbatch*), cancel computing jobs (*scancel*), monitor their existing jobs (*squeue*) or look at the history of previously submitted jobs (*sacct*) that is stored in the *Slurm*

¹²<https://web.archive.org/web/20190306044130/https://hpcc.usc.edu/support/documentation/slurm/>

accounting database. There are also administrative commands that can be used to configure the *Slurm* system (*scontrol*) and to configure the *Slurm* user accounts (*sacctmgr*).

Each *Slurm* computing job has to specify the software or application that must be run and the computing resources that must be allocated to it¹³. The main resources normally specified are the following:

1. Number of CPUs required
2. Amount of RAM required
3. Maximum running time of the computing job (walltime)

Due to Ilifu's many users, many computing jobs (dozens to hundreds) are typically submitted during the course of a day. *Slurm* manages the resource requests by adding them to a queue for pending computing jobs.

2.3.2 Partitions

Since Ilifu has many projects and use cases with different computational requirements, different job queues were created to split up the workloads¹¹. This was done using *Slurm* partitions. A *Slurm* partition is a grouping of computing nodes and can be thought of as a separate job queue².

Ilifu has the following *Slurm* partitions¹¹:

1. **Main:** This is the main partition used on the Ilifu cluster and is generally used for already tested, stable, and highly CPU-intensive batch computing jobs.
2. **Jupyter:** This partition includes the computing nodes dedicated for use with the *Jupyter* web application, which can be used interactively for development, testing and running of analytic workflows.
3. **Devel:** This is the development or testing partition for batch computing jobs. Resources are not allocated to a specific job, but all resources remain shared. This enables faster testing of computing jobs, but makes the pool less stable.
4. **HighMem:** The high memory partition is for computing jobs that

¹³https://docs.ilifu.ac.za/#/getting_started/submit_job_slurm

require a high amount of memory and that cannot be split into multiple smaller computing jobs.

5. **GPU:** This partition is for batch jobs making use of GPUs.
6. **JupyterGPU:** This partition allows the *Jupyter* web application to also request dedicated GPU resources.

2.3.3 Scheduling

Computing clusters like Ilifu must efficiently manage multiple competing jobs from different users and projects. The scheduling mechanisms used to prioritise and allocate these jobs directly impact both the cluster’s utilisation and the experience of its users. In particular for this project, the discussion of the scheduling system is important, since it influences the complexity required for our Slurm simulator. This section examines three aspects of *Slurm*’s scheduling system: the basic First In, First Out (FIFO) queue management, the backfill scheduling mechanism that improves resource utilisation, and the Multifactor Priority system that enables fine-grained control over job prioritisation.

2.3.3.1 FIFO and Backfill

Each *Slurm* partition can have different prioritisation or scheduling rules for its job queues. These rules will likely affect the waiting time of computing jobs in the queue and are therefore relevant for this project. By default *Slurm* uses FIFO (First In, First Out) scheduling for job prioritisation¹⁴. As the name implies, this means that jobs are executed in the order that they are received. However, *Slurm* also provides many more built-in configuration options, as well as plugins to fine-tune the behaviour of its job queues¹⁶. One problem with FIFO scheduling is that large resource-intensive jobs at the front of the queue may have to wait a long time before they can be executed. However, this can have the effect of blocking the execution of many smaller jobs. This generally results in a cluster with a lower utilisation rate and that is less responsive¹⁵. *Slurm* addresses this problem with its backfill scheduling option.

¹⁴https://slurm.schedmd.com/priority_multifactor.html

¹⁵https://slurm.schedmd.com/sched_config.html

In order to explain the backfill scheduling in a more general way, look at the first jobs in the job queue as having higher priority. Similarly then the jobs at the back of the queue will have lower priority. Backfill scheduling enables computing jobs with lower priority to be executed before the higher priority jobs, as long as the starting time of any higher priority jobs is not delayed in the process¹⁵. Since backfill scheduling takes the starting time of higher priority jobs into account, it is important that the backfilled jobs have relatively accurate time limits.



Figure 2.6: An illustration of Backfill scheduling. Three scenarios are shown, each scenario showing the job allocations across time and the CPU resources. The first diagram shows an example of FIFO scheduling where three jobs of different sizes were run in order. The second scenario shown is an example of successful backfill scheduling, where it was possible to execute Job 3 earlier, without delaying any previous jobs. The third example shows an example where backfilling scheduling is not possible, as it would delay the starting time of a previous job.

Figure 2.6 shows an example of backfill scheduling. Using FIFO scheduling, the jobs are executed in order with Job 1 executed first, followed by Job 2 and Job 3. However with backfill scheduling it is possible for Job 3 to be execute first, as long as it does not delay the execution of Job 2. As a result of backfilling, the resources are used more effectively. However, backfilling is also a time-consuming operation to perform. This is due to *Slurm* having to constantly scan the job queue to check if any jobs can be backfilled¹⁵. *Slurm* does have the ability to compensate for this cost by providing multiple configuration options to fine-tune the backfill scheduling for the specific cluster. In Ilifu's case all of these parameters are used with *Slurm*'s default parameters¹⁶ (e.g., performing a backfill scheduling scan

¹⁶<https://slurm.schedmd.com/slurm.conf.html>

every 30 seconds)

2.3.3.2 Multifactor Priority

Even on Ilifu's main partition there are still multiple projects and use cases. Backfill scheduling typically increases the responsiveness for smaller computing jobs, but it does not provide prioritisation between different job types, projects, and users. To address this, in addition to backfill scheduling, Ilifu also uses the Multifactor Priority plugin for scheduling in *Slurm*¹⁷. The impact of this plugin on the waiting time of computing jobs will be part of this research project. The Multifactor Priority plugin makes it possible to rank computing jobs using multiple factors (e.g, job size, job age etc.). The value of each factor value is generally bounded between 0 and 1, with 0 indicating no effect on the priority of a job and 1 having the full effect. The priority value of a computing job is then calculated as a weighted sum of the factor values. The scheduler will choose computing jobs with the highest priority values in the queue to execute first¹⁴. Equation 1 shows an example of such a job priority calculation:

$$\begin{aligned} P = & W_{fairshare} \times F_{fairshare} + \\ & W_{QOS} \times F_{QOS} + \\ & W_{size} \times F_{size} + \\ & W_{age} \times F_{age} + \\ & W_{association} \times F_{association} \end{aligned} \tag{1}$$

The factors and their actual weights used in the Ilifu cluster are discussed below:

1. **Fairshare** ($F_{fairshare}$): This is the most critical factor for Ilifu, promoting equitable access to the cluster¹⁷. The weight $W_{fairshare}$ is set to 100,000, reflecting its importance. The value of $F_{fairshare}$ varies between 0 and 1, with more details in section 2.3.4.
2. **QOS** (F_{QOS}): The Quality of Service factor is used mainly for prioritising interactive jobs¹⁷. Since most Ilifu jobs are batch jobs¹³, the weight W_{QOS} is 20,000. The factor value F_{QOS} ranges from 0 to 1.

¹⁷https://docs.ilifu.ac.za/#/tech_docs/fairshare

3. **Size** (F_{size}): This factor gives priority to larger computing jobs. On Ilifu, it is weighted with $W_{size} = 12,500$. F_{size} reaches its maximum of 1.0 when all computing nodes are requested¹⁴.
4. **Age** (F_{age}): Reflecting the job’s queue time, F_{age} increases to 1.0 after 7 days. The weight W_{age} is 500, ensuring the Fairshare factor remains predominant¹⁷.
5. **Association** ($F_{association}$): Any association (or accounting group) can be assigned an integer priority. This makes it possible to increase the priority of jobs requested by specific accounting groups. However to ensure the association factor value is between 0 and 1, the assigned priorities are normalised by the largest assigned value. On Ilifu, this factor is not used so it is weighted with $W_{association} = 0$.

While this section focuses on Ilifu-specific factors, the Multifactor Priority plugin also supports additional factors not detailed in this report. For the sake of completeness and review, in the next section we expand on the Fairshare factor’s key role in Ilifu’s job priority calculations.

2.3.4 Fairshare

The fairshare factor measures the difference between the portion of computing resources promised and the amount of resources consumed¹⁸. One can therefore ensure that each user receives fair, proportional access to the cluster by adjusting the fairshare factor accordingly¹⁷. However, as will be discussed further in the ‘Shares and Usage’ section below, usage is monitored at multiple accounting levels, as well as individual user levels. In terms of relevance for this project, the influence of this factor in the pending job queue is that computing jobs linked to underused accounts are prioritised over computing jobs linked to overused accounts¹⁴. This implies that on average the waiting time of computing jobs with overused accounts will likely increase and vice versa.

2.3.4.1 Shares and Usage

In addition to astronomy, Ilifu has a few large consortia corresponding to funding groups that each have proportional access to the cluster. This

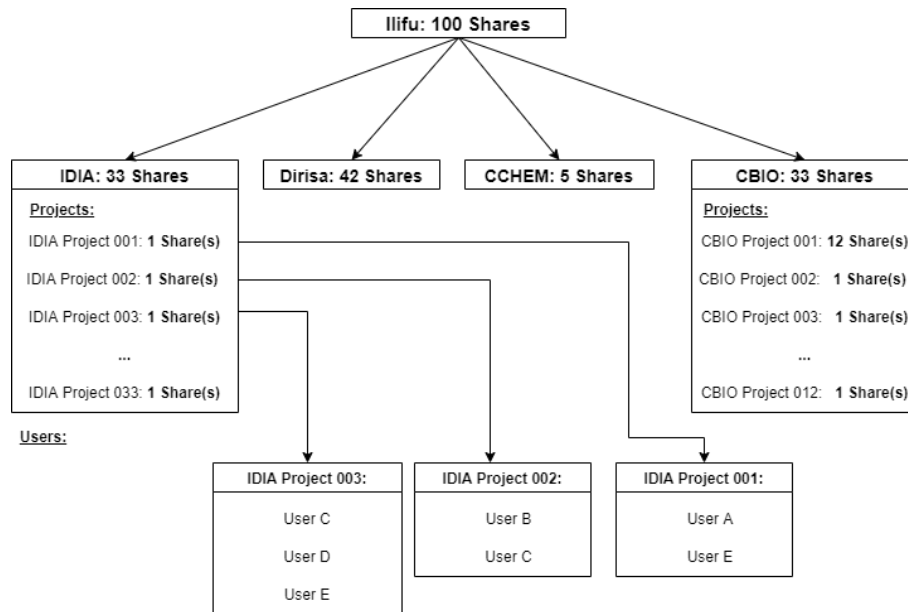


Figure 2.7: An illustration of Ilifu’s share distribution. The diagram shows how Ilifu’s 100 shares are allocated across four consortia: IDIA, Dirisa, CCHEM, and CBIO. It demonstrates the hierarchical breakdown of shares, particularly for IDIA and CBIO, showing individual project allocations. The lower section of the diagram illustrates user assignments for three IDIA projects, detailing how shares are further broken down from the consortium level down to specific users within projects.

proportional access is stored in *Slurm* by shares allocated in a top-down hierarchical fashion to funders, projects and users¹⁸. Figure 2.7 shows a diagram of Ilifu’s share hierarchy. At the top level, the split between the funders is encoded to honour their funding proportions. Thereafter, shares are allocated to individual projects. Projects are generally allocated an equal share, but in some cases an important project can get allocated a higher number of shares (e.g., CBIO Project 001). Lastly, users are assigned to one or multiple projects¹⁷.

The *Slurm* Accounting database, which was briefly discussed in section 2.3.1, is used to store the share information. However, its main use is to store a record of all the computing jobs that were executed on the system¹⁹. This information is used to calculate the resources that were consumed by each level of the share hierarchy from the funder to the user level¹⁴.

2.3.4.2 Fairshare Calculation

The Fairshare calculation ranks users according to the proportion of their allocated shares that have been used. The highest ranked user is given a Fairshare value of 1.0. The ranking of the users is done using the Fairtree

¹⁸https://slurm.schedmd.com/fair_tree.html

¹⁹<https://slurm.schedmd.com/accounting.html>

algorithm. This algorithm ensures a consistent ranking at every level of the hierarchy. Therefore, if two accounting groups are siblings and account group A has a higher fairshare value than account group B, then all of the users of account group A should also have a higher fairshare value than account group B¹⁸.

$$L_{Fairshare} = \frac{S}{U} \quad (2)$$

$$S = \frac{S_{raw_{self}}}{S_{raw_{self+siblings}}}$$

$$U = \frac{U_{raw_{self}}}{U_{raw_{self+siblings}}}$$

To perform this ranking correctly, a *level fairshare* ($L_{Fairshare}$) value is first calculated for each accounting group and user. This calculation is shown in Equation 2. First the shares (S) are normalised by dividing the raw shares of an account (or user) by the total shares allocated to it and its siblings. Similarly, the effective usage (U) is the raw usage normalised by the total usage of the account and its siblings. The *level fairshare* is the normalised shares divided by the effective usage. Users or accounting groups that were overused will have a value between 0.0 and 1.0. Underused accounts and users will have a value larger than 1.0¹⁸.

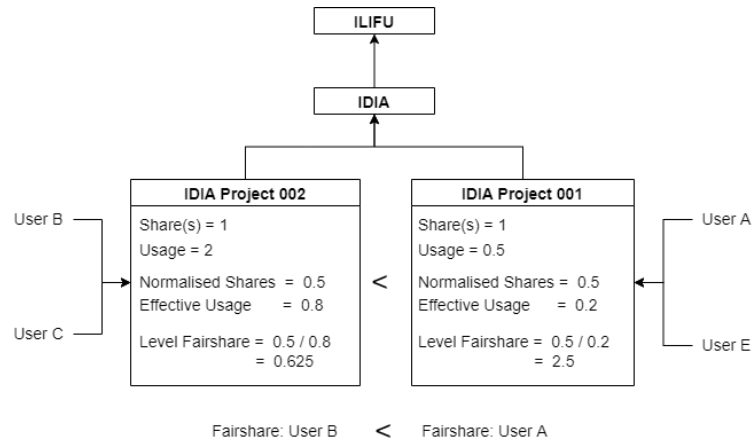


Figure 2.8: An example of Level Fairshare Calculation. The diagram illustrates the calculation for two IDIA projects: 001 and 002. It shows the hierarchical structure from Ilifu down to individual users, with each project allocated equal shares but with different usage levels. The figure details the calculation steps, including normalised shares, effective usage, and the final level fairshare value for each project. IDIA Project 002 demonstrates higher usage, resulting in a lower fairshare value compared to IDIA Project 001, despite equal initial share allocation.

Figure 2.8 gives an example of this calculation for IDIA Project 001 and IDIA Project 002. Both were allocated 1 share of IDIA’s 33 shares of the Ilifu cluster. However IDIA Project 002 has already used 200% of this allocation, while IDIA Project 001 has only used 50% of the allocation. The normalised shares for both are equal to 0.5 ($\frac{1}{1+1}$), while the effective usage for IDIA Project 002 is 0.8 ($\frac{2}{2+0.5}$) and for IDIA Project 001 is 0.2 ($\frac{0.5}{2+0.5}$). Dividing the normalised shares by the effective usage gives a *level fairshare* of 0.625 for IDIA Project 002 and 2.5 for IDIA Project 001. This indicates that IDIA Project 001 has been underused and that IDIA Project 002 has been overused. Since IDIA Project 001 has a higher *level fairshare* than IDIA Project 002, all of its users (user A and user E) will have a higher fairshare value than IDIA Project 001’s users (user B and user C).

We have discussed the challenges and aims of our study, focusing on the Ilifu computing cluster and its scheduling complexities. The next chapter will explore the theoretical foundations that support our methodology. We will review the literature on queue waiting time prediction, tracing its evolution from the basics of queueing theory to the modern use of Discrete Event Simulation (DES) in High-Performance Computing (HPC). This review will not only lay the groundwork for our proposed model but also highlight its relevance within the wider field of computational research.

Chapter 3

Prediction of Queue Wait Time

Building on the introduction chapter’s discussion of the complexities and challenges in the Ilifu computing cluster, this chapter focuses on the theoretical frameworks that have shaped our understanding of queue waiting times. Dating back over a century, the prediction of queue waiting times has been a topic of extensive study (Erlang, 1909), with queueing theory emerging as a pivotal mathematical tool (J. F. Kingman, 2009). As we venture into the intricacies of these theories, we will see how they relate to and inform the methodologies employed in our project, especially in the context of HPC and the innovative use of a Process Equivalent (PE) function. The chapter begins with a fundamental overview of queueing theory, progresses to the application of Discrete Event Simulation (DES), and concludes by reviewing related works that have aimed to predict queue waiting times in HPC environments.

3.1 Queuing Theory

Queueing theory is the mathematical study of queues (Sundarapandian, 2009). It uses probabilistic models, known as queueing models, to represent systems that deliver services (Bose, 2013). These models are designed to facilitate the examination of performance metrics such as average queue waiting time and average queue length (Sundarapandian, 2009). The initial queueing models were developed by Agnes Erlang to model the congestion in telephone networks (Erlang, 1909). The models have since found widespread use in many fields, especially in communications, traffic control and computer networks (J. F. Kingman, 2009). Figure 3.1 shows a

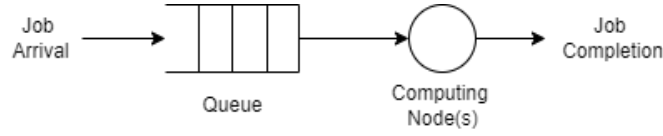


Figure 3.1: A simple queueing model for computing jobs. Jobs arrive, wait in a queue, are processed by computing node(s), and then complete.

model of a queue in an HPC context. The computing jobs arrive at a queuing point and have to wait in a queue if all the computing nodes are busy. Eventually, a job gets allocated to an open computing node, runs to completion and then exits the queueing system (Bose, 2013). In general, a queueing model can be described using Kendall notation, with a popular model being the M/M/1 queue (Kendall, 1953). For this model an analytical formula can be derived for the average queue waiting time, which will be used to verify the simulation model built for this project.

3.1.1 M/M/1 Queue

Figure 3.2 shows a model for the M/M/1 queue. The first M indicates that the arrival process is a Markov process (i.e., memoryless¹), with new jobs received at a rate of λ jobs per time unit. The second M indicates that the service time is Markov as well, with jobs completed at a rate of μ jobs per time unit. The last number indicates that there is one computing node² modelled in the system. The implication of the Markov assumptions implies that the M/M/1 queue assumes the arrivals come from a Poisson process and the service times follow an Exponential distribution (Roy et al., 2021).

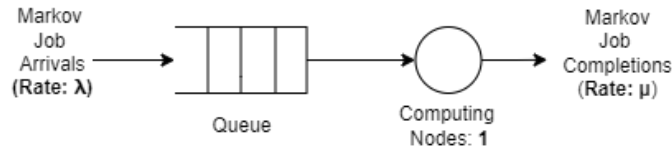


Figure 3.2: A Markov queueing model with arrival rate of λ and completion rate of μ . Jobs arrive, wait in a queue, and are processed by a single computing node.

¹For a stochastic process $A(t)$, the Markov Property means the future states of the process only depend on the current state of the process.

$$Pr[A(t_{n+1}) = a_{n+1} | A(t_n) = a_n, \dots, A(t_1) = a_1] = Pr[A(t_{n+1}) = a_{n+1} | A(t_n) = a_n].$$

(Bose, 2013)

²A computing cluster with multiple computing nodes can also be modelled as a single system, as long as the rates correspond to the arrival rates and completion rates for the whole cluster

One important metric in queueing system analysis, which is relevant in this work, is the system utilisation. It is defined as $\rho = \frac{\lambda}{\mu}$. The metric indicates how hard the computing cluster is working (Stewart, 2009).

The second metric considered in this work is the average waiting time, denoted by W_q . This metric is estimated by $W_q = \frac{\rho}{\mu(1-\rho)}$ and is valid for the case where $\rho < 1$. For $\rho \geq 1$, the queue becomes unstable and the average queue length and average weight time tend to infinity (Bose, 2013).

In an HPC context, in particular, the system often exhibits various inefficiencies (e.g due to job fragmentation) which prevent the system from running at its maximum capacity (Frachtenberg et al., 2005). It is therefore possible for the system to be unstable and for job waiting times to only depend on the length of measurement (i.e., effectively tending to infinity), even at a much lower system utilisation rate. This leads to a third important metric for HPC systems known as the saturation point of the system, which is defined as the "maximum utilisation that can be achieved" before the system becomes unstable. Although measurement can be subjective, a suggestion is to monitor the system (Frachtenberg et al., 2005).

3.1.2 M/M/1 with Priority Queuing

A model of particular interest to this project is the M/M/1 with priority queueing instead of using a FIFO (First-In First-Out) queueing. There exist different studies on priority queueing and the work mentioned here is from Bose (2013). More precisely, a total of P different priority classes are considered, with class P being the highest priority class and class 1 being the lowest priority class. In addition, the priority model is non-pre-emptive, which means that a higher priority job can not stop the active run of a lower priority job (Stewart, 2009). The job arrival rate for each class is assumed to be λ_p and the job completion rate μ_p . The overall job arrival rate is denoted by λ and calculated by summing over all the priority classes with $\lambda = \sum_{i=1}^P \lambda_i$. Similarly, for the overall job completion rate denoted as μ , with $\mu = \sum_{p=1}^P \mu_p$.

It is noteworthy that each priority class has its own system utilisation rate, calculated as $\rho_p = \frac{\lambda_p}{\mu_i}$. The overall system utilisation is then given by $\rho = \sum_{i=1}^P \rho_i$. Similar to the M/M/1 (FIFO) queue, a stable system

requires $\rho < 1$.

The waiting time formulae for the different priority classes can be simplified by defining the variable $R = \frac{\rho}{\mu}$. The waiting time for the highest priority class, denoted by W_{qP} is then given by $W_{qP} = \frac{R}{1-\rho_P}$. The waiting time for any lower priority class, denoted $W_{q(P-i)}$ is then given by $W_{q(P-i)} = \frac{R}{(1-\sum_{j=0}^{i-1} \rho_{P-j})(1-\sum_{j=0}^i \rho_{P-j})}$ for $i = 1, \dots, (P-1)$.

3.1.3 G/G/1 Queue

The previous models were limited to exponential service times and Poisson arrivals, which are often invalid in the context of an HPC system (Frachtenberg et al., 2005). In this context, the G/G/1 queuing model is more applicable. This model is a generalisation of the M/M/1 model for general distributions (indicated with G), which means any distribution can be used for both arrivals (the first G) and for service times (the second G) (J. F. C. Kingman, 1961). It also operates only on one server. It should be noted that inter-arrival times and service times are still assumed to be identical and independently distributed.

Kingman's formula for the approximate mean queue waiting time is given by $E(W_q) \approx (\frac{1}{\mu})(\frac{\rho}{1-\rho})(\frac{c_\alpha^2 + c_s^2}{2})$. The main difference from the M/M/1 waiting time formula is the last term. This term (i.e., $\frac{c_\alpha^2 + c_s^2}{2}$) represents the variability in the system. The c_α refers to the coefficient of variation for arrivals, which is the ratio between the standard deviation of arrival times divided by the mean arrival time ($c_\alpha = \frac{\sigma_\alpha}{\mu_\alpha}$) (Abdi, 2010). Similarly, c_s is the coefficient of variation for service time. The approximation for waiting time is known to be quite accurate, especially in the case of high system utilisation (J. F. C. Kingman, 1961).

It is noted that the focus of this study is on the single server queue models, hence only the relevant metrics related to such models were presented for simplicity. Interested readers are referred to the works of Alvarez et al. (2016), Frachtenberg et al. (2005) and Feitelson et al. (1998) for more advanced analytical models, which account for various features of HPC systems. In the following section, a popular simulation approach often used to model waiting time in the context of HPC is discussed.

3.2 Discrete Event Simulation

A discrete event simulator (DES) is a model that simulates a system as a series of discrete events in time. At each discrete event the state of the system modelled can change, but in between events the systems state remains unchanged. The main assumption made is that the state of the system does not change between two successive events (Stewart Robinson, 2004). These models are commonly used for modelling operational systems including client service systems, healthcare, manufacturing, transportation research and many others (Stewart Robinson, 2004). As previously mentioned, the models are also useful for the study of queueing models. This is particularly true for complex queueing models where the analytical queueing theory results would be intractable or give non-closed-form solutions (Bose, 2013).

3.2.1 Waiting Time Distributions

In the previous section, the formulas shown for the average queue waiting time metric for M/M/1 and G/G/1 queueing models were derived using analytical methods. However, these formulas simplify the distribution of waiting times achieved into a single scalar value. In contrast, when a simulation method is applied with a DES, the whole distribution of waiting times can be estimated. This approach is illustrated in this section by performing a preliminary experiment on a M/G/1 queueing model. Such a model allows general distributions to be used for the job service times. In particular, the effect on the queue waiting time metric is analysed when using a exponential, uniform and normal distribution for job service times. Figure 3.3 shows a diagram of the other assumptions for the queue being modelled. New jobs are assumed to arrive every 2.5 hours, implying a rate of 0.4 per hour. All computing jobs are assumed to be identical, using the full resources of the single computing node and are processed on a FIFO basis. The job service time is set to 2 hours on average, implying a rate of job completion of 0.5. As shown in the diagram, this gives a system utilisation of 0.8. The specific service times distributions used are the following:

1. Exponential distribution with $\lambda = \frac{1}{2}$
2. Uniform distribution with $a = 0$ and $b = 4$

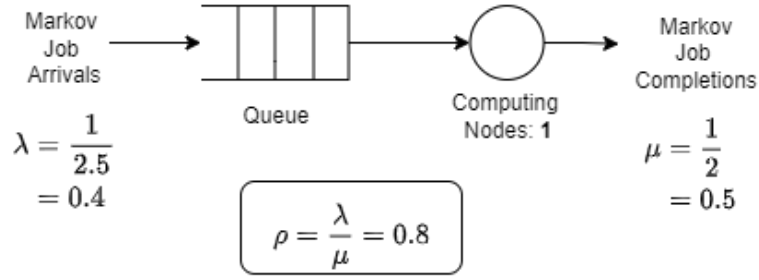


Figure 3.3: A Markov queueing model (M/M/1) with an arrival rate of $\lambda=0.4$ and completion rate of $\mu=0.5$. Jobs arrive, wait in a queue, and are processed by a single computing node. A calculation is also shown for the system utilisation (ρ), which is 0.8.

3. Normal distribution with $\mu = 2, \sigma=0.5$

The *Simpy* Python software was used for the modelling, performing 10,000 job simulations to estimate the waiting time distributions.

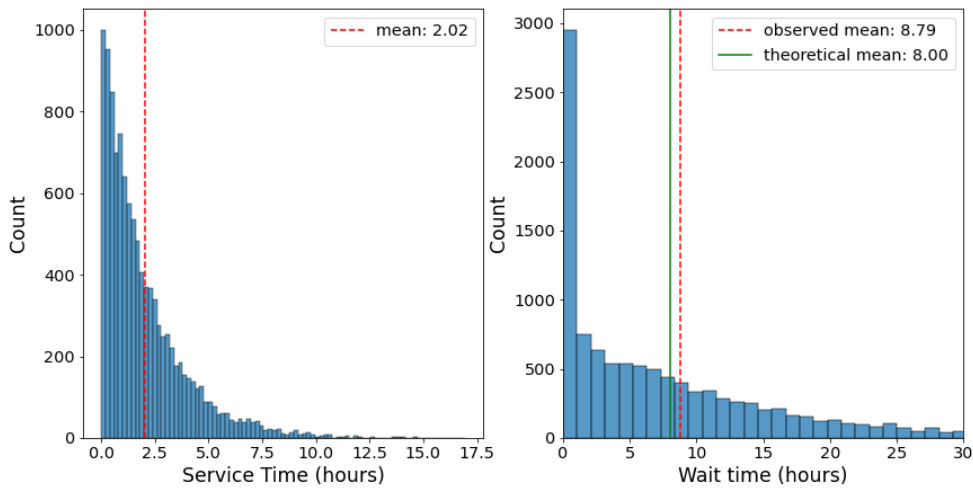


Figure 3.4: These plots show the service time and resulting waiting time distribution for a queueing system with exponential service times, based on 10,000 samples. The left plot shows the distribution of the job service times, with the observed mean indicated by a solid red line. The right plot shows the resulting waiting time distribution, with the observed mean shown as a solid red line and theoretical mean shown as a solid green line. Both distributions exhibit the characteristic long-tailed shape typical of exponential and related distributions in queueing theory.

Figures 3.4, 3.5 and 3.6 show the service time distributions and the resulting simulated waiting time distributions. The mean of each simulated service time distribution is shown to be very close to the target value of 2 hours. The resulting waiting time distribution is also shown, along with the observed and theoretical mean value. Stewart (2009) describes the shape of these waiting time distributions as being "partly discrete and partly continuous". The high peaks at zero correspond to the probability of arriving at the queue and finding it empty (i.e, the discrete part). The rest of the distribution corresponds to the probability of finding the queue

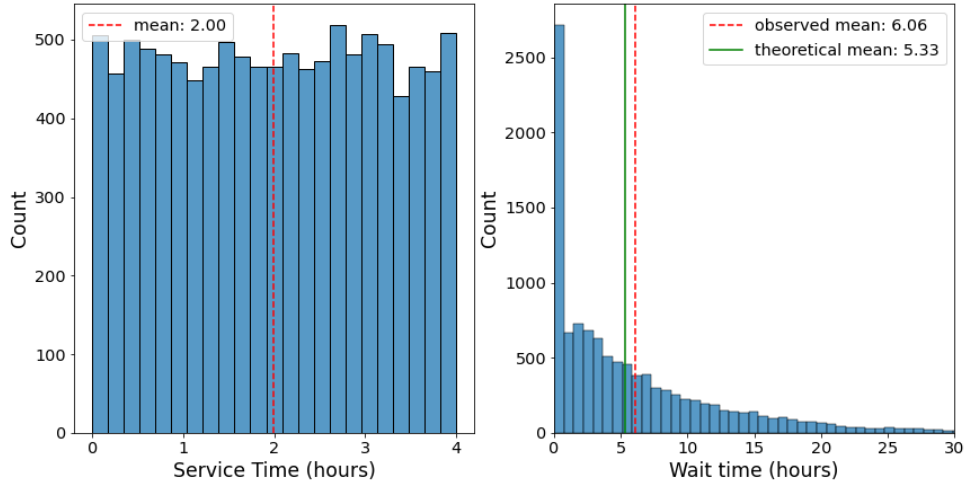


Figure 3.5: The waiting time distribution for a uniform service time distribution. The left plot shows the uniform distribution of job service times, with the mean indicated by a red line. The right plot displays the resulting waiting time distribution, with observed (red) and theoretical (green) mean waiting times shown.

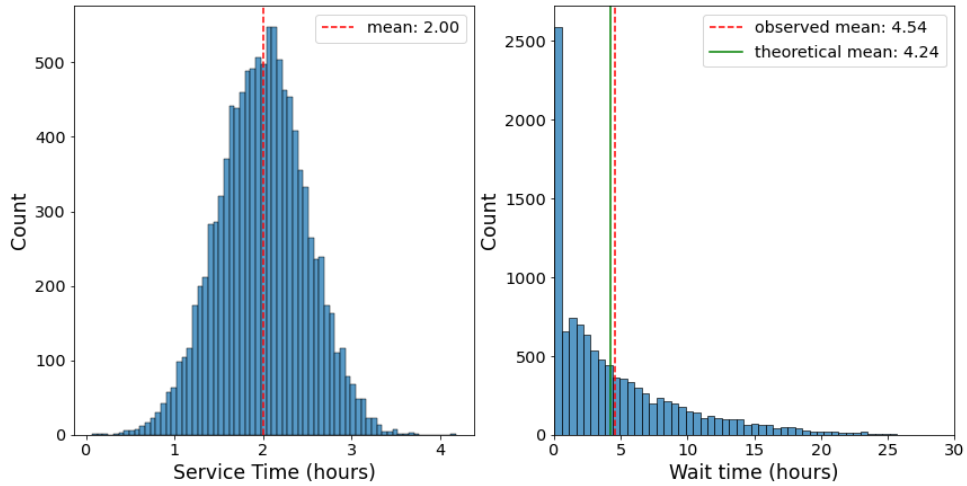


Figure 3.6: The waiting time distribution for a normal service time distribution. The left plot illustrates the normal distribution of job service times, with the mean marked by a red line. The right plot presents the resulting waiting time distribution, including observed (red) and theoretical (green) mean waiting times.

non-empty (i.e, the continuous portion). The exponential distribution has the highest theoretical mean of 8, followed by the uniform distribution at 5.33 and the normal distribution at 4.24. Overall, the average waiting times estimated using the simulation method were relatively close to the theoretical values from analytical methods, even using only 10,000 simulations. In this simple case, the order is a reflection of the variance of each of the service time distributions, with the exponential distribution having the highest variance, followed by the uniform distribution and the normal distribution (Bose, 2013). The benefit of using a DES to estimate

the waiting time metric is that it allows greater flexibility for specifying the processes and assumptions than analytical models, which rely on simplified assumptions and processes. For instance, the analytical queueing models considered above only consider the consumption of a single resource (i.e, the number of servers). However, in an HPC context computing jobs often request different amounts of multiple shared resources (e.g., CPU's and RAM). Extending analytical models to consider multiple resources types is challenging. Interested readers are referred Lim et al. (2012). Schedulers solve the problem through the use of a heuristic functions. The following section will discuss a common heuristic function called the processor equivalent function, which will be used to simplify the modelling of multiple resources in our simulation model.

3.2.2 Processor Equivalent Function

The Processor Equivalent (PE) Function of a computing job is a metric that determines a computing job's most constrained resource on a computing cluster and translates it into an equivalent number of processors requested (Klusáček et al., 2014). The PE function was initially used within the fairshare policy of the Maui scheduler (Jackson et al., 2001).

Previously schedulers would typically only monitor a single resource (e.g. CPU's) in order to adjust the job priorities to improve fairness on a computing cluster (Klusáček et al., 2014).

$$V_{PE} = \max\left(\frac{M_{memory_alloc}}{M_{memory_node}}, \frac{N_{CPUs_alloc}}{N_{CPUs_node}}\right) \times N_{CPUs_node} \times N_{nodes_alloc} \quad (1)$$

Equation 1 shows how to calculate the PE value of a computing job (Jackson et al., 2001). The final calculated PE value is indicated by V_{PE} . M_{memory_node} and C_{CPUs_node} indicate the total memory and CPU resources available on a single computing node. While the resources allocated to the computing job per node are indicated by M_{memory_alloc} and N_{CPUs_alloc} . Finally, N_{nodes_alloc} is the total number of computing nodes allocated to the computing job.

To get a better intuition for the PE calculation, suppose a computing job named *Job 1*, requested 8 processors and 128 GB of RAM on a computing

cluster. Let's also assume the computing cluster had only a single computing node with 32 processors and 256 GB of RAM. The proportion of resources used by the processor request is 25% ($\frac{8}{32}$), while the proportion of resources used by the memory requested is 50% ($\frac{128}{256}$). The memory requested is therefore the most constrained resource of the computing job and expressing it as an equivalent number of processors gives a PE of 16 ($50\% \times 32$). Since only the most constrained resource is used in the PE calculation, it is possible for computing jobs with very different memory and CPU resource requests to have the same PE value. Figure 3.7 shows an example of two such computing jobs. The first computing job shown is *Job 1* which was just discussed. The second computing job requested 16 CPU's and 32 GB of RAM. For this job the CPU's requested is the most constrained resource, using 50% ($\frac{16}{32}$) of the cluster, which also gives a PE value of 16. According to the PE measure, both jobs are comparable, since both used 50% of the resources of the computing node. In a more extreme case, where both jobs had a PE value of 32, meaning they used 100% of the resources of a node it would mean they are exactly equivalent, because no other job would be able to run on the node.

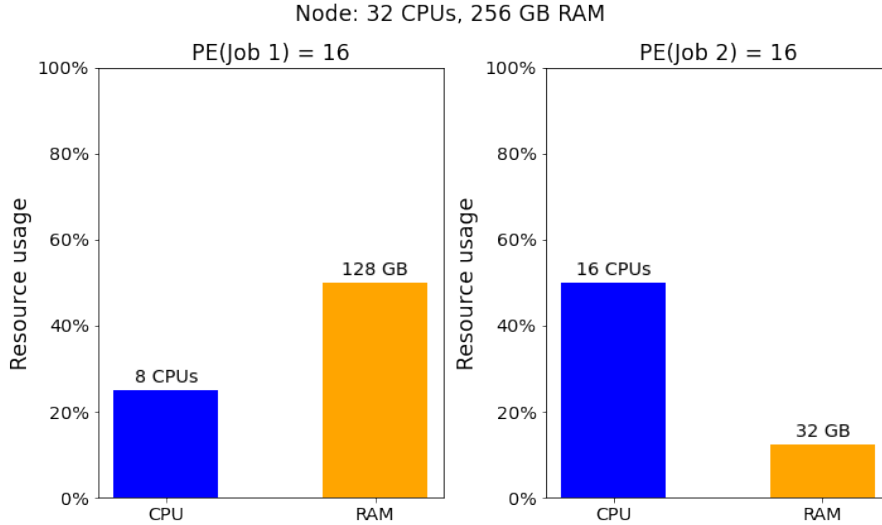


Figure 3.7: An illustration of two different computing jobs with a similar impact on a computing node, even though they have different resource requirements. In the left plot, a job is shown with low CPU usage (shown in blue), but that uses 50% of the RAM (shown in orange) of the computing node. In the right plot, a computing job is shown that uses little RAM, but requests 50% of the CPU's of the node. In both cases the PE value for the jobs would calculate to be 16, which is half of the 32 CPU's available on a node. According to the PE measure, both jobs are comparable, since both used 50% of the resources of the computing node.

Another useful metric is to multiply the PE of a computing job by its running time (in seconds) on the cluster (i.e., $PE_{time} = PE \times T_{elapsed}$). An example is shown in Figure 3.8, where the usage of the Ilifu cluster for 2021 is measured in PE Time units. The results are normalised using 100 computing nodes to show the percentage of cluster usage.

Ilifu Cluster Processor Equivalent Time - weekly
January - December 2021

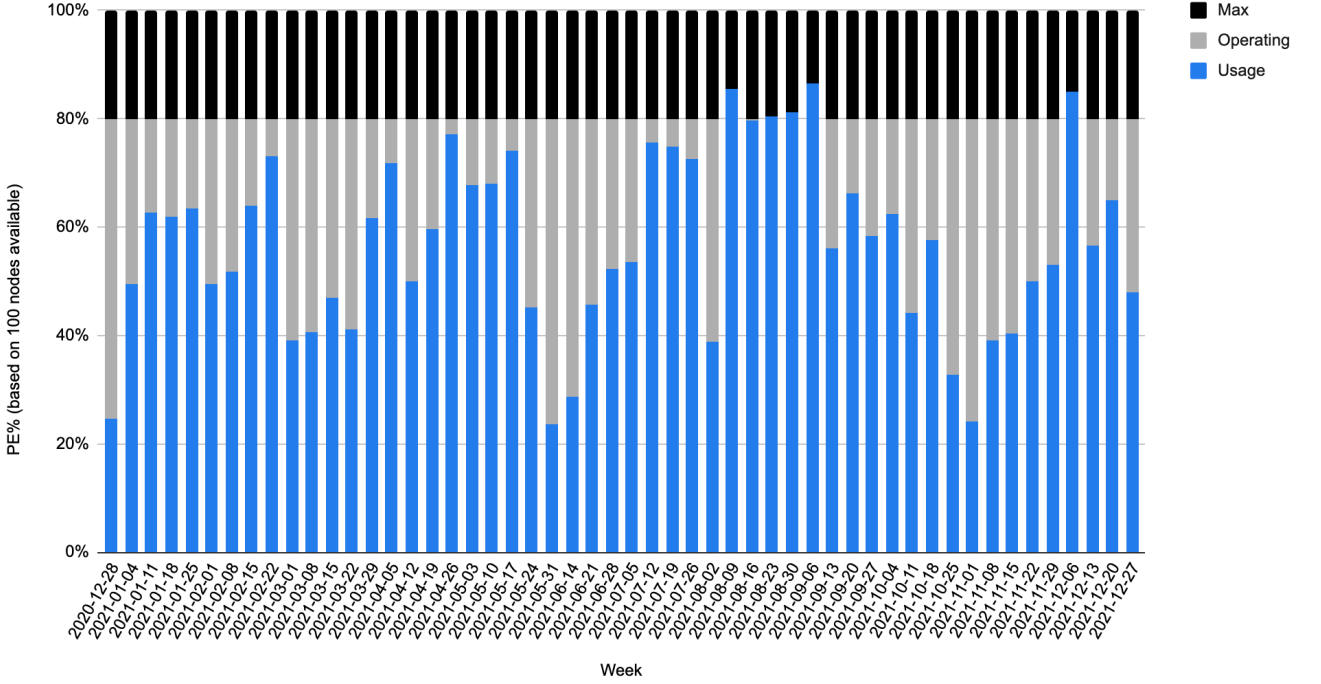


Figure 3.8: The ilifu cluster usage per week in 2021. The black region indicates the reserve capacity on the cluster, while the grey region indicates the normal operating capacity. The blue bars indicate the weekly cluster usage in terms of Processor Equivalent Time units, the values have been normalised assuming a total of 100 nodes in the ilifu cluster. Source: Ilifu 2021 Annual Report.

The busy periods are periods where more than 80% of the cluster was used, as 20% of the cluster is generally left to the cluster administrators and for emergency capacity. The graph shows the variability in weekly usage.

3.3 Related Work

The analytical queueing models discussed in Section 3.1, focused on the mathematical modelling of queues, while the Discrete Event Simulators discussed in Section 3.2, focused on queue simulation. In this section, we will explore the HPC Literature with a focus on simulating HPC queues and predicting the waiting time of jobs. These works are categorised by Murali et al. (2016) as scheduler-specific models for queue wait time prediction as they require the modelling of both the HPC queues and their scheduling policies.

Papers on scheduler-specific models aim to demonstrate that queue waiting time for a new job can be estimated if the actual run times of other

computing jobs in the queue and the scheduling algorithm of the job scheduler are known. Downey (1997) focused on a FIFO scheduler, estimating the queue waiting time for the leading job in a queue. This estimation was based on the remaining run time of active jobs, modelled using a log uniform distribution, and employed mean and median-based predictors for the waiting time estimation.

In contrast, W. Smith et al. (1999) predicted the queue waiting times of jobs by using historic job accounting data for run time predictions. The template method was used for these predictions. A template was defined as a set of job characteristics that could be used to categorise historic jobs into different groups (e.g. queue name, number of nodes). For each group, a runtime could be predicted using different methods e.g. regression or averaging. A genetic algorithm was used to search for the template and prediction method used. The queue waiting times were then estimated by simulating different scheduling algorithms including FIFO, Least Work First (LWF) and backfilling.

H. Li et al. (2004) followed a similar approach to Smith for job run-time predictions. However, instead of just one template, the average was taken over all templates to improve run-time prediction. For the scheduling algorithm, the Maui scheduler was used. The Maui scheduler supports many advanced scheduling strategies, including backfilling, priority sorting and fairshare (Jackson et al., 2001). It also has a built-in simulator that could be used in this case, but it was too slow for real-time waiting time prediction. Instead, the Maui simulator was controlled by a custom program in order to make it skip between events similar to a DES. This speed-up did assume that no dynamic policies were followed e.g. fairshare.

Another noteworthy paper using a DES-based scheduler is Tang et al. (2009). They introduced Qsim, a DES scheduler used at Argonne National Laboratory to simulate the Cobalt job scheduling system’s job queue and resource allocation. Qsim proved useful for scenario testing (e.g., testing different scheduling algorithms), but could also be used to estimate queue waiting times.

Building on these insights, our project proposes to develop a DES-based scheduler for the *Slurm* scheduler, but using a Processor Equivalent Metric for resource modelling.

Chapter 4

Methodology

In the previous chapter, we explored the theoretical foundations of queueing models and discrete event simulation (DES), and ended with the proposal to develop a DES-based scheduler for the *Slurm* system using a Processor Equivalent (PE) metric for resource modelling. In this chapter, we document the development of such a model, which we will refer to as the SimPE (Simulation Processor Equivalent) model. The SimPE model represents a practical implementation of the concepts discussed earlier, modelling cluster resources over time as a single dimension measured in processor-equivalent (PE) units. This approach allows us to compare heterogeneous jobs using a single metric, thereby greatly simplifying the modelling of complex queues with diverse user groups and job requirements. The SimPE simulation model is meant as a proof of concept to show how a processor equivalent simulation framework can be applied to the study of the Ilifu computing cluster. It is generally meant as a tool to gain insight into the busy periods, when the Ilifu cluster has high demand (as shown in section [3.2.2](#)). In this chapter, we first discuss the SimPE model at a high-level, outlining its features and design principles. Thereafter, we discuss the software tools used for its implementation and the key-iterations in the model’s evolution. We conclude the chapter with an in-depth look at the final structure of the SimPE model’s code to provide insight into its implementation and functionality.

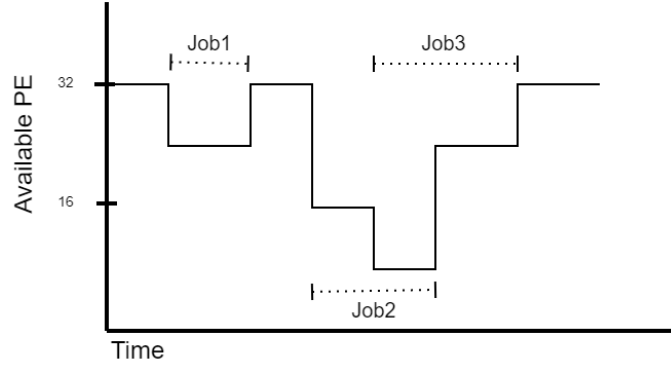


Figure 4.1: This plot shows the available PE as a function of time as jobs are submitted. Three jobs are indicated, the bracketed dashed lines indicate the job durations.

4.1 SimPE Simulation Model

Based on the *Slurm* scheduling policies used on the Ilifu cluster (discussed in Section section 2.3.3) and the modelling of simple queuing models in Section 3.2, we decided on the following design principles for the SimPE simulation model:

1. The available PE of the system should be modelled as a continuous variable over the simulation time.
2. The basic logic for computing jobs should be implemented. This includes the following:
 - Computing jobs should be generated with a start time, service time, amount of PE requested and job priority. For example, Figure 4.1 illustrates a simple case with three jobs with different arrival times.
 - Once received, a computing job should create a request for the amount of resources and enter a pending queue until chosen to be executed next.
 - Computing jobs should only be executed if the full PE requested is available; i.e., partial executions won't be supported.
 - The next computing job to be executed will generally be the one with the highest priority, unless the requested resources are available at the time.
 - Once a computing job is executed, the available PE on the cluster has to be reduced by the job PE. For example, in Figure 4.1, job 1 reduced the available PE by 8 when it started execution.
 - The simulation model needs to keep track of the computing jobs

being executed (referred to as the active queue)

- Once a computing job completes execution, the resources used should be released. For example, in Figure 4.1, job 1 increased the available PE by 8 when it completed execution.
3. It should support backfilling of jobs. This enables smaller PE jobs to be executed before bigger PE jobs, if the capacity is available.
 4. It should support dynamic priority sorting of jobs, e.g. changing of priority during a job's life cycle. This will assist in the modelling of some of the factors on Ilifu that affect the priority of queued jobs e.g. a user's cluster usage or how long a job has been queuing.
 5. The Simulation model should support the running of a batch of jobs (e.g., trace), so it is possible to compare results to the Ilifu cluster.
 6. The Simulation model should be fast to enable multiple runs to be done in quick succession. This will allow the analysis of the distribution of metrics.

Finally, SimPE should be able to capture accounting information about the simulation. This should include the starting time, waiting time and ending time of each job simulated.

4.2 Software

In order to implement the SimPE model, we considered the different discrete event simulation packages that are available in the Python ecosystem. The two main packages we decided to consider were the *Simpy* and *Salabim* packages. We chose to evaluate these packages specifically due to its capabilities and overall feature set, but also because of their active development communities and the general quality of their documentation. The main concern that was kept in mind was the ease with which we would be able to model the computing cluster and its scheduling policies in our PE-based model. In the following section we will give a summary of our experience using both of these models to implement the SimPE model. This will include a short history about each package, their strengths, limitations and suitability for our project.

4.2.1 Simpy

*Simpy*¹ is arguably the best known discrete event simulator software package for Python (Matloff, 2008). In *Simpy* processes are modelled using Python generator functions (Huling et al., 2015). A process can be any active portion of the system being modelled, for example a customer in a queue, a cashier at a till or a computing job in a queue (Matloff, 2008). *Simpy* is ideal for basic and conventional use-cases in discrete event modelling (Huling et al., 2015).

Simpy also has built in classes that model the bottlenecks or shared resources in a system such as the resource class and container class². The resource class enables the modelling of a resource that can only be used at the same time by a limited number of processes – for example the number of CPUs available on the cluster². The container class on the other hand, enables the modelling of the level of a resource (either discrete or continuous) and allows the level to be decreased or increased². This was used to model the Processor Equivalent units available on the cluster.

The main advantages of using *Simpy* for this project include the following:

1. Succinct expression of simulation models (Huling et al., 2015). The shared resource classes especially assist with the succinct expression of commonly used models².
2. Fast prototyping of new simulation models (Huling et al., 2015).

The disadvantages of using *Simpy* for this project are:

1. The shared resource classes do not support all the features required for this project. In particular, there is no support for priority queueing in the container class². Unfortunately, the internals of the shared resource classes - including the container class - are not very accessible (e.g., the pending queues or active queue can't be accessed directly). It is therefore not possible to implement priority queueing manually by modifying the queues directly. The implementation of these features would thus be much harder and require creating our own shared resource classes, based on the existing classes.

¹<https://simpy.readthedocs.io>

²https://simpy.readthedocs.io/en/latest/topical_guides/resources.html

2. There is no built in support for monitoring for the shared resource classes. Any detailed monitoring will therefore require adding monitoring code dynamically at runtime to the existing classes or implementing our own versions of the classes that include monitoring³.

4.2.2 Salabim

*Salabim*⁴ is a Python package for discrete event simulation (Ham, 2018). It was developed by Ruud van der Ham starting in 2016⁵ as a response to changes in the *Simpy* ecosystem. The development of *Salabim* was motivated by the transition from *Simpy* 2.0 to *Simpy* 3.0. While *Simpy* 3.0 aimed to simplify the package by removing many features, this simplification came at the cost of backwards compatibility. Many sophisticated and advanced simulation models based on *Simpy* 2.0 could not be easily ported to version 3.0. This situation led to the development of *Salabim*, which can be seen as an enhanced version of *Simpy* 2.0 that avoided these compatibility issues while introducing new features.

The main advantages of *Salabim* over *Simpy* version 3 - in terms of ease of modelling - are the addition of the features⁵, also referred to as process interactions⁶, that are listed below.

1. Activate: A *Salabim* process has a few possible states. Each process starts off in an inactive state and has to be activated before it becomes live.
2. Hold: A *Salabim* process can be paused until a specified time.
3. Passivate: This function pauses a *Salabim* process for an indefinite period of time.

Salabim also has many helper classes that can be used for statistical sampling, queue handling, shared resources, monitoring and for real time animation of the process (Ham, 2018).

The advantages of *Salabim* for this project include the following:

³https://simpy.readthedocs.io/en/latest/topical_guides/monitoring.html

⁴<https://www.salabim.org>

⁵<https://www.salabim.org/manual/About.html>

⁶<https://www.salabim.org/manual/Component.html>

1. Support for priority queuing for continuous resources (like the PE available on the cluster). *Simpy* only supports priority queues for discrete resources.
2. The ability to access and change the pending and active queues of resource objects. This makes it possible to introduce more complicating prioritisation mechanisms. As a result, dynamic prioritisation models can be developed, which mimic the operation of real world clusters.
3. The inclusion of a monitoring system: For the built-in classes, the monitoring system captures variables that would be important for built-in classes e.g. duration of time in queue, length of queue. Custom variables can also be added to the simulation. In particular, the distinction made between non-level monitors and level monitors leads to straightforward and transparent capturing of the important simulation values.
4. Documentation: The documentation is very comprehensive and has many modelling examples.

The disadvantages of *Salabim* for this project include:

1. The coding style is more verbose than *Simpy*, since a class is generally defined for every process. This will lead to more complicated models.
2. *Salabim* makes some implicit assumptions about the prioritisation of jobs, that were not applicable to this project. A workaround had to be found to resolve this. This is discussed in detail in *Salabim*: FIFO section.

4.3 Key Model Iterations

In this section, we discuss the key model iterations for the development of the SimPE model. These iterations represent key functionality milestones in developing a more realistic cluster scheduler. This included first implementing a proof of concept model for FIFO scheduling using the *Simpy* package. Thereafter, the FIFO scheduling model was reimplemented in *Salabim*, where it was expanded to handle jobs with static priority values. We also discuss the specific reasons for making the switch to the *Salabim*

model. The final two biggest model iterations were adding support for dynamic job priorities, where the priorities of jobs can change over time, and adding support for job backfilling, which helps to increase cluster utilisation. For each of these model iterations we show a process diagram and discuss the nature of the changes made to the model.

4.3.1 Simpy: Proof of concept model

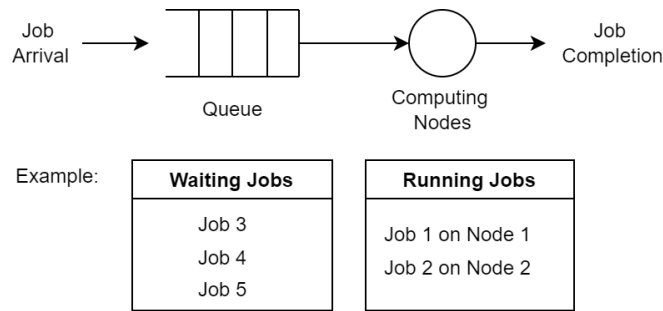


Figure 4.2: An illustration of the FIFO job model that was implemented with *Simpy*. The top diagram shows the general FIFO model, with jobs arriving, queueing and executing on computing nodes. The bottom diagram gives a practical example of five jobs that arrived in order, with the first two jobs busy executing and the remaining three jobs waiting in the queue in the order of their arrival.

As stated above, the initial model was implemented using the *Simpy* package. The first iteration of this model assumed a first-in first-out (FIFO) queue with a constant job inter-arrival time and constant job service time. Figure 4.2 shows a flow diagram of the FIFO model and the flow of jobs from arrival to completion. An example is also shown of five jobs that arrived in order, with the first two jobs (job 1 and job 2) busy executing and the rest waiting in the pending queue in the order they will be executed.

This model was then extended to include exponential job inter-arrival times as well as testing the effect of simple random distribution for job service times (e.g. uniform, normal, exponential). The amount of PE requested by jobs was also changed to vary randomly from a list of possible values.

The above modelling was done using the *Simpy* container class to keep track of a continuous resource (i.e., remaining PE of the cluster) over the course of the simulation. However this class only supports simple FIFO queues, and the real Ilifu cluster has an advanced prioritisation between jobs. *Simpy* does have a class that supports some prioritisation called the resource class, but this functionality only supports discrete values. *Salabim* on the other hand supports more advanced prioritisations. We find that

Simpy does not offer the required functionality for our use-case, but it is a useful starting point to explore basic FIFO queues. In the next section we explore the use of *Salabim* for our Simulation model.

4.3.2 Salabim: FIFO and Static Priority Queuing

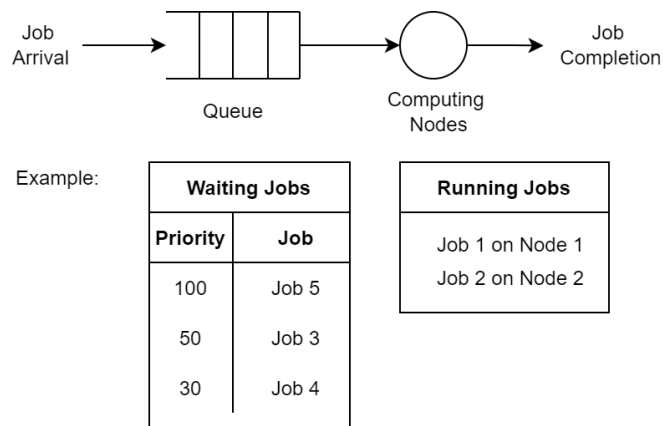


Figure 4.3: An illustration of the static priority job model that was implemented with *Salabim*. The top diagram shows the generic queueing model with jobs arriving, queueing and executing on computing nodes. The bottom tables give a practical example of five jobs that arrived, with the first two jobs busy executing and the remaining three jobs waiting in the queue, but with a queue that is ordered by their priority values instead of their arrival time.

Figure 4.3 illustrates a static priority model and the flow of jobs from arrival to completion. A job arrives, is included in the pending queue, the job with the highest priority score is executed first and finally the job completes. An example is also shown of five jobs. Note that the waiting job queue is ordered by priority instead of job arrival.

All of the models up to this point simulated the jobs directly. However when performing test runs against the Ilifu cluster it is necessary to run a specific batch of jobs. A batch mode was built into the model, allowing the running of a specific batch of jobs, being read in a specific order.

4.3.3 Salabim: Dynamic Job Priority

For the dynamic job priority model, we aimed to remove the simplifying assumption of priorities remaining static over the duration of a job. Instead, we wanted the priorities of jobs to be more similar to *Slurm*'s Multifactor Priority plugin, which was discussed in Section 2.3.3.2. In particular, the support of multiple priority columns, as well as the ability for some priority columns to change over time.

In order to achieve this, the static priority model was upgraded in two stages. First, support was added for multiple priority columns with different sorting directions. Thereafter, the second stage was to add support for a priority columns to change over time.

Figure 4.4 shows an illustration of the first stage of this upgrade process, namely the multiple column priority model. It also includes an example of five jobs moving through the model. As can be seen from the table

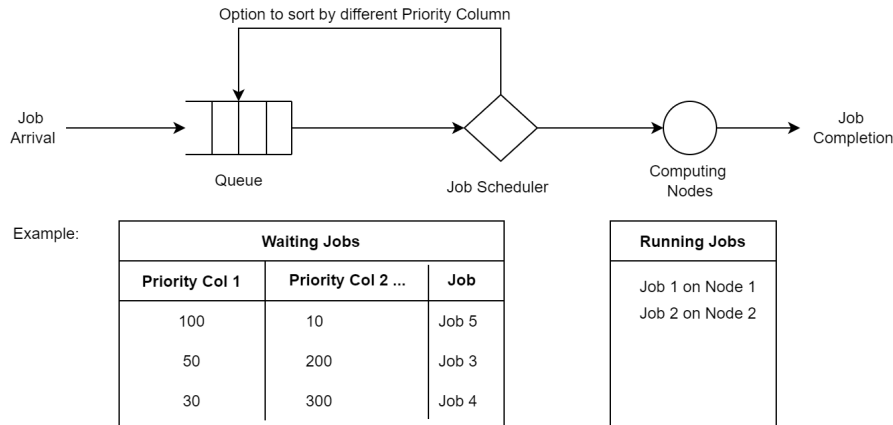


Figure 4.4: An illustration of the static priority job model that was implemented with Salabim. The top diagram shows jobs arriving and then queueing. Thereafter, a job scheduler process is shown (indicated by a diamond), which is able to change the order of the queued jobs by sorting by one or a combination of the Priority columns (e.g., to match Slurm a weighted sum could be calculated over the Priority columns). Thereafter, the highest priority job is executed on the computing nodes when the resources become available. The bottom tables gives a practical example of five jobs that arrived, with the first two jobs busy executing on the cluster and the remaining three jobs waiting in the queue, but with their multiple priority values shown.

of waiting jobs, each job now has two priority columns instead of one. Additionally, sorting by Priority Column 2 would give a very different ordering to Priority Column 1. Although it would not make a difference for this example, the aim of stage 1 was to be able to sort by multiple columns simultaneously. An example of where this would be useful is when one column was the job's priority, while the other was the job's arrival time. In this case, it would be better sorted for the highest priority first, and thereafter for the lowest job arrival time.

Unfortunately, the built-in *Salabim* queue that was used to store the pending jobs did not support having multiple priority columns. In order to get around this limitation, the pending queue was extracted into a Python *pandas*⁷ dataframe. This process was repeated whenever the SimPE model was required to prioritise jobs. A separate column was created for each job field, including columns for `job_name`, `job_pe`, `job_priority`, `job_runtime` and `job_entertime`.

⁷<https://pandas.pydata.org/>

In order to give control over the sorting process, two parameters were created in the SimPE model, namely *sort_columns* and *sort_asc*. The *sort_columns* variable was a list of the columns used to sort the pending job queue. *sort_asc* variable was a list of boolean values indicating the direction (i.e., True for ascending) of each *sort_columns* column.

4.3.4 Salabim: Backfilling

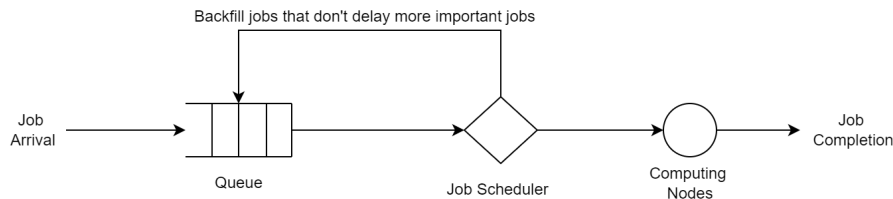


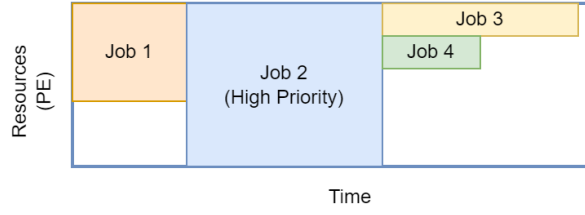
Figure 4.5: An illustration of a backfill job model that was implemented with Salabim. The diagram shows jobs arriving and then queueing. Thereafter a job scheduler process is shown (indicated by a diamond), which is able to change the order of the queued jobs. If there are free resources available on the cluster and jobs that can be backfilled without delaying other higher priority jobs, then these jobs are allowed to jump the queue and execute on the computing nodes.

The Ilifu cluster allows the execution of jobs with lower priority, so long as the jobs don't delay any higher priority jobs. This is job backfilling and ensures that the cluster is used as effectively as possible without delaying important jobs. *Salabim* does not have built-in support for backfilling, but some of its classes are useful for the implementation of the process. In the Simulation model, when it is time to start executing the next job, instead of just running the highest priority job directly, a backfill process is first followed. This process is illustrated in Figure 4.6 and outlined below.

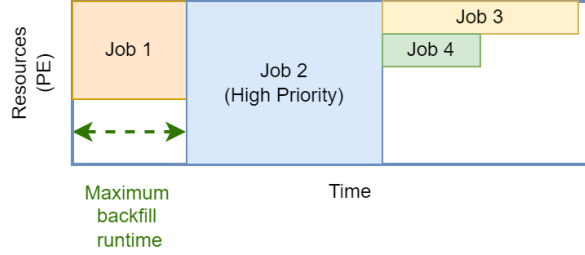
The following algorithm is assessed immediately upon job submission:

1. Build a job schedule: Use the active jobs being executed and the pending jobs waiting for execution to build a job schedule. This is a table that shows the start and ending time of each computing job as well as the available PE on the cluster available at each point.
2. Calculate the maximum backfill runtime: Calculate the longest time available at the current PE, until the current job is blocked by a higher priority job. This process is then repeated for each job in the pending job queue.
3. Pick the highest priority job to backfill and reduce the available PE on the cluster.
4. Repeat step 3 until no jobs remain that can be backfilled.

1. Build Job Schedule



2. Calculate: Maximum backfill runtime



3. Backfill where possible



Figure 4.6: An illustration of the backfill calculation process. Three diagrams are shown to breakdown the process, with each diagram showing the job allocation in the dimension of time and PE resources. In the first diagram, four jobs are shown, with some available space currently available which would allow jobs to be backfilled. In the second diagram, the same jobs are shown but with an additional distance measure (referred to as the 'Maximum backfill runtime') is calculated to quantify how long the current available resources would be available for the purposes of backfill scheduling. In the third diagram backfilling occurs according to the implementation in the model.

4.4 Modelling Detail

When building the SimPE model using the *Salabim* discrete event simulator, we started by modelling the number of PE units available on the cluster. As previously mentioned in Section 4.2.2, the built-in resource class in *Salabim* already met many of our requirements, since it supported a continuous amount of resources, as well as priority queueing. We created a resource named *res*, and gave it a capacity equal to the amount of PE units on the cluster. The Python code that was used to set up the *res* resource, as well as other resources and queues used in the SimPE model is shown in Listing 4.1.

In order to replicate the FIFO queueing behaviour of our initial *Simpy* model, we were required to use a second resource object. We named this resource *ticket* and gave it a capacity of 1. It was used as a gateway to

ensure that there could only be one concurrent request of the *res* resource. In addition to creating an extra resource, we also created two *Salabim* queue variables, namely *pendingQ* and *activeQ*. As the names imply, the *pendingQ* stored a sorted list of jobs waiting to be executed. Likewise, the *activeQ* stored a sorted list of jobs currently running on the cluster. Instead of defining separate queues, it would also have been possible to use the internal queues used by the *res* or *ticket* resources for tracking resource requests and usage. However, for this project using separate queues seemed to be beneficial, since it made it possible to be more explicit about when jobs entered or exited the queues.

```

1 import salabim as sim
2
3 # Parameters
4 CLUSTER_PE_AVAIL=8
5
6 # Resources
7 res = sim.Resource("res", capacity=CLUSTER_PE_AVAIL)
8 ticket = sim.Resource("ticket", capacity=1)
9
10 # Queues
11 pendingQ = sim.Queue("pendingQ")
12 activeQ = sim.Queue("activeQ")

```

Listing 4.1: Python code for setting up the Queues and Resources for the SimPE model

When modelling a queuing system with *Salabim*, it is broken down into a few key processes and each is modelled separately⁸. For each process, a new class is made based on *Salabim*'s *Component* class. The behaviour of the process is specified within the *process* method of the new class. An example of such a class definition and accompanying *process* method can be seen in Listing 4.2. To aid with readability in this section, we will refer to such a new class as a process, and make it explicit when referring to the *process* method.

Before explaining how the SimPE model was modelled using processes, it would be helpful to understand how they can be used. Firstly, as previously explained in Section 4.2.2, a process can be in one of a few different states in *Salabim*. The two main states are the *data* state and *active* state. The *data* state means the process does not have a running *process* method, and therefore will remain static throughout a simulation. In contrast, the *active* state means the process does have a running *process* method. The *activate* method can be used to turn a process from *data* to *active*.

⁸<https://www.salabim.org/manual/Overview.html>

The other states that are relevant for this project are the *current*, *scheduled* and *passive* states. The *current* state means the process is currently being executed by *Salabim*. The *scheduled* state, means the process is scheduled to execute at a specific future time. Lastly, the *passive* state, means the process is scheduled to happen at time infinity (in other words never). The main methods to change the state of process into the *scheduled* state is the *hold* method. While the main method to change a process into the *passive* state is the *passivate* method. Lastly, the *activate* method can be used to turn a *passive* state into the *current* state.

In the case of the SimPE model, three separate processes were used, namely the *JobReader* process, the *Job* process and the *Slurm_scheduler* process. When using the SimPE model to run a simulation, the main input was expected to be a job batch file. This file would contain multiple jobs and all the information required for each job's simulation on a cluster. The *JobReader* process would use this information and create new *Job* processes in the SimPE model at each job's respective arrival time. Thereafter, the *Job* processes would keep track of the each job's details, and make it possible to track the state of the each job on the cluster (i.e., queueing, running or completed). It would also specify what happened when a job transitioned between different states (i.e., newly created to queueing, queueing to running or running to completed). The *Slurm_controller* process then determined which computing job to run next on the cluster, following the scheduling policy selected for that simulation run.

The modelling of each of the three processes in the SimPE model required the following columns of information for each job in the batch file:

1. **Job Arrival time:** The time of job submission in in date-time format, up to the closest second.
2. **Job Number:** A number mainly used for job identification.
3. **Job PE:** The resources requested by the job, but measured in PE Units.
4. **Job Priority:** A number indicating the priority of the job, with a higher number indicated a higher priority job.
5. **Job Run time:** The total run time of a computing job in seconds.

The *JobReader* process can be summarised in a few lines of Python code, which is shown in Listing 4.2. The first three lines show how the *Salabim* package was imported, and that the *JobReader* class was based on

Salabim's *Component* class⁹. This is the main class used in *Salabim* to model the key processes of a simulation and usually most of the logic is specified in the *process* method of the class. In line four, the *JobReader* created its own process method and also added an input parameter, namely *job_batch_list*. Thereafter, in line 3, the code looped through every job in the *job_batch_list*. In order to delay each job until their arrival time, we used the *hold* function with the *job_entertime* (aka arrival time) as a parameter. This function had to be called using the Python *yield* statement, and paused the process until the specified simulation time was reached. Finally, the new *Job* processes were created at the correct arrival time, using the job's specifications from the job batch file.

```

1 class JobReader(sim.Component):
2     def process(self, job_batch_list):
3         for job_spec in job_batch_list:
4             yield self.hold(till=job_spec.job_entertime)
5             Job(job_spec=job_spec)

```

Listing 4.2: Python code for the *JobReader* process of the *SimPE* model

The code for the *Job* process is shown in Listing 4.3. First, a *setup* method is defined in line 2, which is the initialisation method that is first called for any newly created *job*. It uses the *job_spec* dictionary as an input variable and sets up the fields that the *job* will keep track of. In particular, this is the PE, priority and run time of the job. However, in addition a *slurm_comment* field is also created. This field is used to capture additional information about the scheduling of a job. For instance, if the *Slurm_scheduler* chooses to backfill a job, this information is captured in this field.

The *process* method of the *Job* class is split up into three different job transitions, which are indicated by the comments in line 9, 17 and 25. In the first section, a job is in transition from a newly created job to a pending job. In line 10, the job was added to the *pendingQ*, indicating that was waiting to be executed. Notably, the priority value used to enter the queue is first multiplied by -1. This is due to *Salabim*'s queue class using lower numbers to indicate a higher priority, whereas *Slurm* uses higher numbers.

After entering the queue, the *job* checks that the *slurm_controller* process is in an active state and scanning for the next job to execute. This is necessary since the *slurm_controller* was not modelled to run periodically, and instead simplified to only run when jobs arrive or complete their runs.

⁹<https://www.salabim.org/manual/Component.html>

Finally, in line 15, the *job* used the *passivate* method to enter a *passive* state.

In the second section, the *job* is in transition from pending to running on the simulated cluster. This means that the *slurm_controller* had chosen to execute the *job* next and therefore used the *activate* method to wake up the *job*. The *job* can now request resources to run on the simulated cluster. However, as seen in line 18, it first requests a single resource (or ticket) from the *ticket* resource.

As previously explained, this ticket resource prevents more than one job from requesting resources concurrently from the main *res* resources. It is only after the ticket request is fulfilled, that it allowed to continue. In line 19, it finally requests PE units from the main *res* resource, with PE being the size of the *job*. Thereafter, the *ticket* resource is released in line 20, thus allowing other jobs to request resources from the simulated cluster. The reason for using the *ticket* resource, is that on *Salabim* it was found that FIFO scheduling worked slightly differently than on the initial *Simpy* proof of concept model. Specifically, it was found that small jobs were sometimes able to bypass larger jobs in the FIFO queue and get executed before them.

An explanation was found¹⁰ for this this behaviour, as well as the remedy of using a *ticket* resource. The cause was that the *Salabim* resource requests only got recognised when the total amount of resources were available on the cluster. Therefore, smaller jobs are able to pass larger jobs in the queue while the larger jobs resource requests have not been recognised. After receiving the resources required for the job, the *job* enters the *activeQ* on line 22. It then simulates the execution of the job by using the *hold* to pause the *job* process for the full run time of the job. In the last section (line 25), the job has completed running. It therefore leaves the *activeQ* and releases the cluster resources. Finally, it activates the *slurm_controller* so that it can use the released resources to find the next job to run. Simplified code for the *slurm_controller*, as well as the setup code required by *Salabim* were provided in Appendix A.

```
1 class Job(sim.Component):
2     def setup(self, job_spec):
3         self.pe = job_spec.job_pe
4         self.prio = job_spec.job_prio
```

¹⁰The *Salabim* Google group had a discussion about the [queue jumping behaviour](#). Note that membership to the group is required to view the link, although anybody can apply.

```

5     self.run_time = job_spec.job_run_time
6     self.slurm_comment = ""
7
8     def process(self):
9         # Job Transition 1: New to Pending
10        self.enter_sorted(pendingQ, -1*self.prio)
11
12        if slurm_controller.ispassive():
13            slurm_controller.activate()
14
15        yield self.passivate()
16
17        # Job Transition 2: Pending to Running
18        yield self.request(ticket)
19        yield self.request((res, self.pe))
20        self.release(ticket)
21
22        self.enter(activeQ)
23        yield self.hold(self.run_time)
24
25        # Job Transition 3: Running to Completed
26        self.leave(activeQ)
27        self.release(res)
28
29        if slurm_controller.ispassive():
30            slurm_controller.activate()

```

Listing 4.3: Python code for the Job process of the SimPE model

Chapter 5

Testing and Implementation

In order to better understand the SimPE model’s accuracy and limitations for our specific use case, it was important to identify representative job types that would effectively demonstrate the model’s behaviour within practical hardware and time constraints. Through careful analysis of our available resources and simulation requirements, we developed a set of job configurations that could meaningfully test different aspects of scheduling behaviour while remaining computationally feasible. By running identical workloads (i.e., the exact same batches of jobs) on both simulation and real systems, we could assess how well the SimPE model’s predicted waiting times aligned with actual system behaviour and understand its practical limitations. This was made possible by creating a separate *Slurm Training Cluster* on the Ilifu infrastructure where batches of jobs could be submitted. Its set-up and configuration are discussed in Section 5.1. However, one disadvantage of the *Slurm Training Cluster* was that the computing jobs ran in real time, while on the SimPE model, they ran significantly faster. Therefore, a second Slurm System was also used to assist with the development and testing of the SimPE model, namely the Slurm Simulator (Nikolay A Simakov et al., 2017). The Slurm Simulator is discussed in detail in Section 5.2, including reasons for its speed-up as well as its limitations. The assumptions used for the job batches of the simulation runs are discussed. This includes the global assumptions such as the job inter-arrival times and the average job execution times, which are discussed in more detail in Section 5.3. The specific assumptions that were made for each simulation run are discussed in Section 5.4. We conclude the chapter with a summary of the selected job configurations that was used for the test simulations runs for the different scheduling types.

5.1 Slurm Training Cluster

A *Slurm Training Cluster* (or *SlurmTrain*) was created on the production Ilifu cluster to perform test simulation runs against the SimPE model. A possible alternative that was considered, was to instead use the existing Ilifu Slurm system for the testing runs. However, it was important to test the SimPE model for the different scheduling types (e.g. FIFO, Backfill, Priority). Unfortunately, this would require the Slurm configuration files to be changed, which can not be done on the production Slurm system without affecting all of the users of the system. In contrast, the *Slurm Training Cluster* provided full control of the Slurm configuration files, as well as control over all the computing jobs running on the Slurm system.

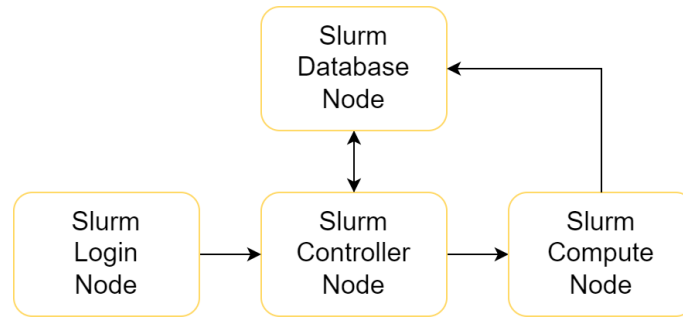


Figure 5.1: A diagram of the Slurm Training Cluster, with the four nodes making up the cluster shown as rectangles and a simplified version of the communication protocols between the nodes indicated with solid lines and arrows. The four nodes include the Slurm login node, Slurm controller node, Slurm database node and Slurm compute node. The Slurm login node is used to submit computing jobs to the Slurm controller node, which handles the scheduling and allocation of resources to computing jobs. Once a job is allocated resources, the Slurm controller node initiates the job on the Slurm compute node. The Slurm controller node is also in constant two-way communication with the the Slurm database node, logging information about the jobs received and their state as well as fetching the information required for scheduling jobs. Lastly, once a computing job finishes execution, this information is also logged in the Slurm database node.

The *Slurm Training Cluster* was made up of four separate computing nodes. Figure 5.1 illustrates these nodes, as well as their implied communications. The Slurm login node was used to access the cluster from outside of the network. It was also used to submit computing jobs to the Slurm controller node. The Slurm controller node received these computing jobs, and oversaw the whole Slurm infrastructure. It stored and retrieved job accounting information from the Slurm database node, as well as allocated jobs to Slurm computing nodes¹.

It is important to note that only a single computing node was used in the *Slurm Training Cluster* and therefore for all test simulation runs. Even though the resources are ostensibly downscaled and represent a tiny portion of the Ilifu cluster, this should have little impact on the applicability

¹<https://slurm.schedmd.com/overview.html>

or relevance of our results, since PE is a normalised or scaled quantity. It also helped to reduce the impact on the production cluster, since a computing node that was allocated to the *Slurm Training Cluster* had to be removed from the production cluster. The specific computing node² used had 8 CPUs and 64 GB of RAM, which is approximately a quarter of a standard Ilifu computing node, which has 32 CPUs and 250 GB of RAM.

The main configuration file for Slurm is called *slurm.conf*. This file contains most of the Slurm configuration options and has been included in full in Appendix B. We will just show an extract of two configuration settings here, that were important for this project, since they affect the way resources are modelled in Slurm.

```
1 # RESOURCES
2 SelectType=select/cons_res
3 SelectTypeParameters=CR_CPU_Memory
```

Listing 5.1: Resource selection settings in *slurm.conf* for the *Slurm Training Cluster*

The default resource modelling for Slurm is to allocate a whole computing node to a computing job. However, for this project we required the ability to allocate a single CPU core and a specified amount of memory to a computing job. The configurations lines in Listing 5.1 changes the default configuration, so that we can do this. This means that a single computing node with 8 CPUs, could then be modelled using with SimPE using 8 PE units. It also enabled us to run the simulations, where multiple resources were requested by each job (CPU cores and memory)

After completing the general configuration of the *Slurm Training cluster*, we added dummy users to the cluster, which would be used to submit Slurm jobs. Since it was unclear how many user accounts would be needed for the test simulation runs, we added a total of 15 users to the cluster, named User-01 to User-15.

In order to make it possible to run simulation runs where jobs had different priority levels, it was necessary to create different accounting groups and assign the users to them. We created two such accounting groups, with the first accounting group called “training”. We assigned User-01 to User-10 to this accounting group, and assigned a base priority of 5,000 to the group. This group was used for normal priority job submissions. The

²The computing node used was flavour type *E*, while a standard computing node on Ilifu is flavour type *G*. The different computing node flavours on Ilifu is documented at <https://docs.ilifu.ac.za/#/openstack/openstack>

second accounting group called “crucial” was used for high priority job submissions and was assigned a base priority of 10,000. We assigned User-11 to User-15 to this accounting group.

It is important to note that the base priorities assigned to these accounting groups, or associations (as they are referred to in the Slurm documentation) are only used when Slurm’s multi-priority plugin is enabled. When the plugin is enabled, these priorities impact the *association_factor*, but only in a relative way. So in actuality the result of these configuration changes were that the *association_factor* was set to 0.5 for the “training” accounting group, and to 1.0 for the “crucial” accounting group. For further details about how this factor was used in this project, please refer to the priority simulation runs, which are described in Section 5.4.2.

At this stage of the process the *Slurm Training cluster* was running and completely configured. However, we still needed to have an automated way of submitting the batches of jobs to the cluster. This was performed by multiple Python scripts that read in a list of jobs from a job batch. For each job this included details about the time when the job should be submitted, the user submitting the job and the amount resources requested. The scripts then waited until the allocated time (or time delta between successive jobs) and submitted the jobs as using the designated user’s permissions.

Finally, we set up systems to monitor the Slurm Training Cluster and to log all the details required for this project. First the configuration files (*slurm.conf*) and the accounting database was stored after each test simulation run. During each simulation run, we also captured the state of the Slurm queue by using the *squeue* command, as well as the job priority information with the *sprio*. This was done by using a scheduled task that runs every minute (i.e. a cron job).

5.2 Slurm Simulator

The *Slurm Training Cluster* worked well as a final testing model for the SimPE model, since it enabled any batch of computing jobs to be run on both models using any Slurm configuration. When comparing runs on the Slurm Simulator against the SimPE model, we ran exactly the same batches of jobs on both systems. This practice was always followed when

comparing two different models. The *Slurm Training Cluster* did, however, have the disadvantage that computing jobs ran in real time, which meant the testing runs could take a long time to run. This was not ideal when the SimPE model was initially developed, since a faster iteration time would ease the development of the model. Therefore, to address this, Slurm Simulators were explored, notably those by Nikolay A Simakov et al. (2017) and Jekanovic et al. (2018). Both of these simulators reuse the open-source Slurm source code but make modifications to the source code, to enable a Slurm computing cluster and its scheduling processes to be simulated at higher than real-time speeds. There were other Slurm simulators also mentioned in the literature, including Lucero (2011), Trofinoff et al. (2015) and Rodrigo et al. (2017). However, according to Nikolay A Simakov et al. (2017) these three simulators had slow simulation speeds even for small clusters, and therefore they weren't considered for this project.

This project opted to use the Nikolay A Simakov et al. (2017) simulator, hereafter referred to as the Slurm simulator, instead of the Jekanovic et al. (2018) simulator, even though the later was documented to have advantages in terms of simulation accuracy. The reasons were mainly pragmatic, since the documentation of the Slurm simulator, assisted more with the compilation and installation process, as well as the usage of the simulator. In contrast, the Jekanovic et al. (2018) simulator had less documentation and was slightly more challenging to compile and get started with. For instance, it required a C programming script to be modified in order to read job trace files. The Slurm simulator was deemed to be sufficient for this project, since it had high simulation speed and was only used to assist with the development of the SimPE model.

In order to use the Slurm simulator to run simulation runs for this project, it had to be set up correctly³. Figure 5.2 gives an overview of the main inputs the simulator requires, as well as the main outputs it generates. In terms of inputs, the simulator requires a Slurm configuration file (i.e., *slurm.conf*) and Simulator configuration (i.e., *sim.conf* and *users.sim*) to be specified. The same slurm configuration file used for the *Slurm Training Cluster* was also used for the Slurm simulator, with only minor alterations. The Slurm simulator also provided a utility to check the slurm configuration file to ensure that the necessary settings were set for the simulator to work. The users configuration file was easy to configure and contained

³https://ubccr-slurm-simulator.github.io/docs/slurm_sim_manual_v1.2.html

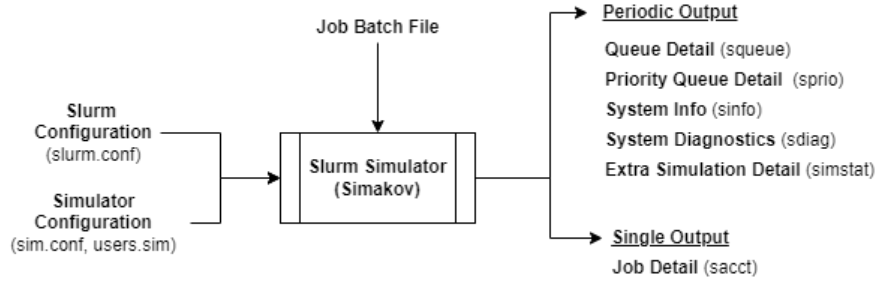


Figure 5.2: A flow diagram of the Simakov Slurm simulator showing the input files required by the model and the generated output files. The input configuration files are shown on the left side of the diagram, with configuration files split into two categories, either relating to the Slurm configuration or to the Simulator configuration. The names of the main configuration files for both categories are shown in brackets. The top of the diagram shows the input data file required by the simulator, which is the Job batch file with the details about the computing jobs for the simulation. The output generated by the simulator is shown on the right side of the diagram, this is split into two categories as well, namely Single Output or Periodic Output. The Single Output files are generated only once at the end of a simulation, for example the job accounting details from Slurm’s `sacct` function. Periodic Output files are generated periodically throughout the simulation and appended to a text file with a timestamp. The diagram shows a description for each type of periodic output generated, with the corresponding Slurm command generating the output shown in brackets. In addition to the periodic output from Slurm commands, the simulator also captures extra simulation statistics (in a file called `simstat.out`), which includes details about the Slurm scheduling loops and backfill scheduling loops performed throughout the simulation.

the 15 usernames generated for the *Slurm Training Cluster*. The most important file to configure before running the simulator was the simulator configuration file (`sim.conf`). This file included all the simulator specific configurations, including the location of the job batch file (`JobsTraceFile`). One important parameter that should be mentioned is the *SpeedScale* parameter. This parameter slows down the backfill scheduler of the Slurm simulator to be more in-line with the speed at which the backfill scheduler runs on Slurm. For this project we used a *SpeedScale* parameter of 5.0, which is the recommended value for small cluster in the documentation³. However, further tuning of this parameter could increase the accuracy of the backfill scheduling of the Slurm simulator.

The Slurm simulator captures outputs similar to those from the *Slurm Training Cluster*, as shown in Figure 5.2. An example is the output of job details, which uses Slurm’s `sacct` utility to output details (e.g. requested resources, submit time, start time, end time etc.) about the jobs that were ran. This output is then saved as a single output file, and is particularly important for this project since the submit time and start time of jobs are used to calculate the job waiting times. The simulator also periodically captured other outputs from Slurm functions, which the typical time interval being every 60 seconds. Among these are the job queue details (`squeue`), which greatly aided in debugging the differences found between the SimPE model and the Slurm Simulator. Additionally, the cluster state information (`sinfo`) provided details about the state of the nodes on the cluster; however, since only one worker node was used, this function was

not particularly useful for this project.

The information about the job priorities (*sprio*) was especially important to determine the abilities of this simulator. Initially it was assumed that the model would support priority scheduling, and it seemed to calculate the priorities of jobs correctly. However, after performing a few test runs it became clear that the scheduling was performing only FIFO scheduling or FIFO scheduling with backfill enabled, depending on the configuration used. The model was therefore mainly used using these two scheduling modes.

The general diagnostic information (*sdiag*) included statistics about the main scheduling loop run by Slurm. It also included information about the amount of backfills since Slurm last restarted, which was useful for the backfill testing runs. The last output generated by the Slurm simulator was statistics about the simulation (*simstat*), which included details about the scheduler and backfill scheduling loops and how often and when they ran in the simulation.

Lastly, it should be noted that a new Slurm simulator was released in 2022 (Nikolay A. Simakov et al., 2022). This simulator is mentioned here for completeness. It followed a new approach to address the shortcomings of the previous model and even enabled the simulator to model dynamic scheduling policies like priority and fairshare scheduling. However, even though the abilities and accuracy of this model was reported to have improved, these came at the cost of speed. The new simulator was not used for this project, mainly due to the timing of its release. However, the reduction in speed of simulation would also make it less useful for this project.

5.3 Job Batch Assumptions

Multiple job batches were created to perform the test simulation runs between the SimPE model, *Slurm Training Cluster* and Slurm Simulator. Some simplifying assumptions were made when creating these job batches. The main assumption was that the job inter-arrival times and the job running times had the Markov property, similar to the M/M/1 model discussed in Section 3.1.1. Both therefore followed exponential distributions. This section will describe the practical considerations that constrained the

parameters and type of exponential distributions that were used for these assumptions. The process used for tuning the system utilisation for a simulation run will also be discussed.

5.3.1 Job Inter-arrival Times

In order to simplify the testing process, our Python script for submitting jobs to the *Slurm Training cluster* was timed using the system time. As a result, the timing of the job submission and the associated *Slurm* job scheduling, was discretised (i.e., in seconds) rather than continuous. This also made comparisons with the Slurm Simulator model easier, since its job trace file format also only supported job inter-arrival times measured in seconds. The distribution used for the job inter-arrival time was an exponential distribution, but with times rounded to the nearest second.

In order to avoid the simultaneous submission⁴ of jobs, the simulation runs were adjusted. The job inter-arrival times were adjusted to have a minimum difference of 1 second between jobs, but to also maintain the targeted mean job inter-arrival time for a test simulation run. As a note, while simultaneous submission of jobs is common practice, implementing this would obscure our understanding of the effect of backfilling and job prioritisation.

The mean job inter-arrival time for each test simulation run was selected to achieve specific system utilisation levels, as discussed in Section 5.3.3.

5.3.2 Job Run Times

Selecting appropriate assumptions for job run times involved two key steps: choosing a distribution and determining its parameter values. We opted for the same shifted exponential distribution used for the job inter-arrival times to maintain consistency and minimise run errors. For instance, initial tests on the *Slurm Training Cluster* with standard exponential assumptions for job run times led to execution errors. These were traced back to jobs having zero-second run times, and consequently, a job wall time (the maximum duration a job is allowed to run) of zero seconds, which essen-

⁴Simultaneous job submission as a dimension of testing was seen as out of the scope of the current study

tially indicates no limit. To address this, for our test simulation runs, we set each job’s wall time by rounding its known run time up to the nearest minute. For example, a job with a 70-second run time was assigned a wall time of 2 minutes.

After settling on the distribution for job run times, we focused on the mean value of the distribution. This assumption was critical across most project simulation runs. The training cluster executed each job in actual wall-clock time. This meant that simulations with longer mean run times would take proportionally longer to complete. As a result, using longer run times could make it impractical to conduct enough simulation runs for statistical validity. Conversely, a shorter mean run time could significantly influence job scheduling due to the Slurm scheduling algorithms’ resolution (typically, backfilling jobs are planned with a one minute resolution). This presented a delicate balance between practicality and accuracy in our simulations. After conducting several test runs, we selected a mean run time of 30 seconds for the shifted exponential distribution. This duration strikes a balance between computational efficiency and the capability to effectively test different Slurm scheduling dynamics.

5.3.3 System Utilisation Tuning

As previously mentioned in Section 5.3.1, the job inter-arrival times were not constant, but were instead a key factor in tuning the system utilisation rate for each simulation run.

The system utilisation (ρ) represents the ratio between the rate at which jobs arrive and the rate at which the system can process them (Stewart, 2009). More specifically, it is calculated as the average arrival rate of jobs divided by the average service rate. When $\rho < 1$, the system can maintain a stable queue as it processes jobs faster than they arrive. However, when $\rho > 1$, the queue will grow indefinitely since jobs arrive faster than they can be processed. This metric was important for this project, since a too high system utilisation rate for a simulation run could cause the queue to become unstable and the waiting times to inflate (Bose, 2013). While different waiting times between scheduling models could still be compared even with high system utilisation, this would amplify the observed differences. This amplification would be particularly noticeable in job batches with a limited number of jobs, potentially exaggerating the

relative performance of different scheduling approaches. In order to reduce the probability of this occurring, we aimed for a system utilisation rate of 60%. Since it still allowed high enough waiting times to test the Slurm scheduling behaviour. At the same time, it was on the conservative side, especially since jobs of different sizes and priorities could increase the system utilisation based on random fluctuation, or for their priority group.

In order to tune the job inter-arrival rate, we estimated the system utilisation rate for each test run using the following formula: $\rho \approx \frac{\lambda}{c\mu}$. The average job arrival rate is indicated by λ , and the average job completion rate is indicated by μ . This is similar to the system utilisation formula discussed in Section 3.1.1 (i.e., $\rho \approx \frac{\lambda}{\mu}$), but adjusted for the fact that multiple jobs can run on the server concurrently. The average number of jobs that could be run concurrently on the system was indicated by c .

There were two main system utilisation tunings that were performed for this project. The first tuning was aimed at the job types used by the FIFO simulation run (Section 5.4.1), priority simulation runs (Section 5.4.2) and multi-resource simulation runs (Section 5.4.6). It assumed that on average four jobs could run concurrently on the *Slurm Training Cluster*, each with a mean run time of 30 seconds. Using the formula specified above, this results in a completion rate of $0.1\bar{3}$ ($c\mu = 4 \times \frac{1}{30}$). A job inter-arrival time (i.e., $\frac{1}{\lambda}$) of 12.5 seconds then gives a system utilisation of 60%. This was rounded up to the nearest second, and a job inter-arrival time of 13 seconds was used for these simulation runs.

The second tuning performed was aimed at the Backfill simulation run (Section 5.4.5). It also assumed that on average four jobs could run concurrently on the Slurm Training Cluster, but each job ran with a mean run time of 45 seconds. Using the formula specified above, this results in a completion rate of $0.0\bar{8}$ ($c\mu = 4 \times \frac{1}{45}$). A job inter-arrival time (i.e., $\frac{1}{\lambda}$) of 18.75 seconds then gives a system utilisation of 60%. This was rounded up to the nearest second, and a job inter-arrival time of 19 seconds was used for these simulation runs.

5.4 Test Simulation Runs

The following sections describe the specific job configurations designed to test different aspects of the SimPE model’s scheduling behaviour. Each

configuration was carefully chosen to examine particular scheduling features while working within our hardware and time constraints. Rather than attempting to test every possible job combination, we focused on identifying representative job types that could effectively demonstrate the model’s behaviour for each scheduling approach. This targeted testing strategy allowed us to thoroughly examine key scheduling features while keeping the test runs computationally feasible.

5.4.1 FIFO

In order to test the SimPE model against the Slurm Training Cluster for FIFO scheduling, we created ten independent job batches to run on both models. Unlike the theoretical models considered in Section 3.1, where the jobs were all identical, we had to include jobs with different resources requests. However, having too many different job types, would make it harder to analyse the results of the simulation runs, especially for smaller job batch sizes. Instead, the project’s scope was limited to only create test simulation runs with two distinct job types. Table 5.1 shows the chosen job types of the FIFO simulation runs along with their main assumptions and frequencies. We opted for having a low resource job type (i.e., 1 PE job) and a higher resource job type (i.e., 3 PE job). A high PE job type of 3 PE⁵ was chosen since it still allows multiple large jobs to run simultaneously on the 8 CPU worker node.

Table 5.1: Job types used in FIFO simulation runs.

Job PE	Mean Runtime (sec)	Frequency
1	30	50%
3	30	50%

In total each job batch had 200 jobs, which gives an expected number of 100 jobs for the 1 PE and 3 PE jobs (i.e, $50\% \times 200$). In general, the simulation runs of this project were limited to have around 100 jobs per job type. This was mainly done to limit the total running time of the ten independent simulation runs, since jobs on the *Slurm Training Cluster* ran in real time. For instance, for the FIFO simulation runs this limited the total time of the ten independent runs to be close to 20 hours. For

⁵A more natural choice for a high PE job might have been to use a 4 PE job since it is a multiple of two. However, the 3 PE job was preferred since the same job types were going to be used for the backfill scheduling runs, and two 3 PE jobs still left 2 CPUs available on the cluster which could be utilised by backfilled jobs

simulation runs with more job types this running time would be longer, but in general simulation runs were limited to be no more than 48 hours.

When setting up the Slurm configuration for this simulation run, the *SchedulerType* was set to use the built in scheduler (i.e, *sched/builtin*). The built-in scheduler is Slurm’s FIFO scheduler and therefore it executes computing jobs in the order that they arrive in. If the resources is not available to execute the next job, then it waits until the resources becomes available before continuing with the queue⁶. This was the only configuration line that was specifically set for this simulation run, however the full Slurm configuration file used is shown Appendix B.

5.4.2 Priority

For the testing of the priority scheduling type there were two main simulation runs that were done. The static priority simulation runs kept the priority of jobs constant throughout the run. While the dynamic priority runs added factors that changed the priority of jobs throughout the run. This section will describe the factors that were chosen for these runs, as well as the way they configuration changes performed on the *Slurm Training Cluster*.

5.4.3 Static Priority

For the static priority simulation runs, we used the same basic job types as for FIFO scheduling (i.e., 1 and 3 PE), but in addition a job could be of a low priority or a high priority. The resulting four job types are shown in Table 5.2. Notably, low priority jobs were given a priority value of 5,000 and high priority jobs a value of 10,000. For static priority runs, the exact values were less important than the priority class, as jobs could not increase priority over time. However, the numeric priority values were important for the dynamic priority values, where the priority values of job does change over time and these were only the starting priority values.

For each simulation run we used a total of 500 jobs. The increase in jobs is to compensate for the fact that there are now four job types instead of two. This results in 125 jobs per priority class on average.

⁶<https://slurm.schedmd.com/slurm.conf.html>

Table 5.2: Job types used in Static Priority runs.

Job PE	Priority Type	Priority Value	Mean Runtime (sec)	Frequency
1	Low	5,000	30	25%
3	Low	5,000	30	25%
1	High	10,000	30	25%
3	High	10,000	30	25%

The main change in the FIFO scheduler configuration was to enable the multifactor priority plugin (i.e., *PriorityType* = *priority/multifactor*). In order to implement the static priority weights, we created two different associations on the *Slurm Training Cluster*, namely *training* and *crucial*. All low priority jobs were run with dummy users that were allocated to the *training* association, while high priority jobs were run with dummy users that were allocated to the *crucial* association. As only the association factor should impact the job priority for these simulation runs, we set all the other multi-priority factor weights equal to zero. Therefore, the job priority formula that applied was the following:

$$\begin{aligned}
P &= W_{association} \times F_{association} \\
&= 10000 \times F_{association}
\end{aligned}$$

As can be seen in the formula, the association factor weight ($W_{association}$) was set equal to 10,000. The association factor ($F_{association}$) for the *training* and *crucial* associations, was set equal to 0.5 and 1.0 respectively. This explains the calculation resulting in the 5,000 priority value for low priority jobs and the 10,000 priority values for high priority jobs seen in Table 5.2.

5.4.4 Dynamic Priority

For the dynamic priority simulation runs, the aim was to use a simple configuration where the priority values of jobs change over time. The main dynamic factors used on the Ilifu cluster are the *Fairshare* factor and the *Age* factor. However, as the *Fairshare* factor was deemed as out of scope for this project, the *Age* factor was used instead for the dynamic priority simulation runs. The *Age* factor prioritises jobs that have spent a long time waiting in the pending job queue compared to the configurable maximum length of time (i.e., *PriorityMaxAge*)⁷. Like the other factors, it is added together in a weighted sum to give a computing job a multi-priority score (see Section 2.3.3.2 for more details about the calculation).

⁷https://slurm.schedmd.com/priority_multifactor.html

The configuration of the static priority runs were used as a starting point for these simulation runs, but with the *age* factor added. This meant that computing jobs already had a starting priority value of either 5,000 or 10,000. In order to counteract these values, the *Age* factor weight (W_{age}) was set equal to 15,000. The resulting job priority equation is the following:

$$\begin{aligned} P &= W_{association} \times F_{association} + W_{age} \times F_{age} \\ &= 10000 \times F_{association} + 15000 \times F_{age} \end{aligned}$$

The *Age* factor (F_{age}) can take on a real value between 0.0 and 1.0. The specific value is proportionate to the length of time that a computing job has been in the pending queue up to the maximum age parameter⁷. In order to set up the simulation run in such a way that the *Age* factor could have an impact on some of the jobs, it would have to be able to change the priority of some low priority jobs (i.e, 5,000 priority jobs) that have a long queueing time, to be higher than that of new incoming 10,000 priority jobs. The result would be that these jobs would then take priority over the incoming 10,000 priority jobs, which would otherwise not have been the case. Since the expected runtime for jobs was low (i.e. mean of 30 seconds), the maximum age parameter was set to only 3 minutes. This means that that any computing job that spent a minute in the queue would gain a priority of 5,000 points (i.e. $\frac{1}{3} \times 15\,000$). It also gave an additional 2 minutes before the *Age* priority value would plateau. This required another parameter to be changed, namely the multipriority recalculation frequency (i.e., *PriorityCalcPeriod*). By default the priority values are every 5 minutes, however this was decreased to the minimum value of 1 minute to ensure the *Age* priority value was recalculated frequently enough to affect some pending job's priorities.

5.4.5 Backfill

On Slurm, backfill scheduling can be applied in addition to the other scheduling types like FIFO or priority scheduling. It is also possible for a batch of job to be executed, without a single job being backfilled. In order to design a simulation run for backfill scheduling it was therefore important that enough backfills occurred to have a noticeable effect on the waiting time of the computing jobs.

A few initial test runs using FIFO scheduling were performed using the Slurm Simulator model with similar job types (i.e., 1 PE and 3 PE jobs)

and system utilisation levels of approximately 60% (i.e. $\rho \approx 0.6$). However, the amount of backfills observed were too low to have any impact on the waiting times, with many runs producing no backfills. The amount of backfills in these test runs proved to be sensitive to the specific sizes and average run times of the job types chosen for the simulation runs. Instead of further complicating the process, it was decided to only focus on the testing of the FIFO scheduling type as base. Even though the SimPE model used the Priority scheduling type and it would be best to test this functionality as well, due to the additional complexity this was deemed out of scope for this project.

Focusing on only testing Backfill runs with FIFO scheduling also had the added benefit of making it possible to use both the Slurm simulator and Slurm Training Cluster. Further investigation of the Slurm Scheduling runs showed that at 60% system utilisation, the runs only had an average pending queue length of 2 jobs. This indicated that a likely reason for the low number of backfills observed was that the short queues observed did not give any chances for backfills to occur, since backfills can only occur in the pending queues of the cluster. However, we did not want to increase the target system utilisation of 60% for these runs directly, since higher levels could easily make the waiting times become unstable, as was discussed in Section 3.1.1. Therefore, instead of increasing the system utilisation of the whole batch, we added a small fraction of 8 PE jobs to backfill simulation runs. These jobs were expected to create a bottleneck, which would temporarily increase the system utilisation during a portion of the run. Since the queues would be longer during these time periods it would create more opportunities for jobs to be backfilled.

Table 5.3 shows a breakdown of the job types that were used for the backfill simulation run. Only 1% of the introduced jobs were 8 PE jobs. Every backfill job batch was made up of 200 jobs, so this amounts to an average of only two 8 PE jobs in a batch of job batch. In addition to this change,

Table 5.3: Job types used in Backfill runs.

Job PE	Mean Runtime (sec)	Frequency
1	30	50%
3	60	49%
8	30	1%

the average running time of 3 PE jobs was set to 60 seconds, instead of 30 seconds. This was done to further improve the probability of job backfills

occurring, since for a job to be backfilled the resources (e.g. CPU or RAM) has to be unused by the cluster for the whole length of the running time of the job. With this change the 1 PE jobs now had a shorter running time than the 3 PE jobs, and it was expected that this would increase the chance that 1 PE jobs would be backfilled in front of the 3 PE jobs.

A few small test runs were performed using the *Slurm Training Cluster*, investigating the number of backfills produced, and these showed that the number of backfills were much higher than the initial test runs using only 1 PE and 3 PE jobs with a 30 second average runtime. However, this did not address the problem of system utilisation, which would be expected to increase with these two changes. To address this, the job interval between jobs was tuned to get to a system utilisation level closer to 60%. As was discussed in Section 5.3.3, the result of the tuning exercise was a job interval of 19 seconds.

When determining the Slurm configurations to use for these simulation runs, the aim was to bring *Slurm*'s backfill procedure more in-line with the simplified version implemented in the SimPE Model. The SimPE model's backfill implementation mainly focused on determining which jobs to backfill. However, the algorithm was still applied at every job event (e.g. job arrival, job completion). By contrast, *Slurm*'s backfill algorithm is applied periodically, by default every 30 seconds. To make it easier to compare the models, we attempted to choose the Slurm backfill configuration parameters that would reduce the effect of this difference in approach between the models.

The main configuration parameters that were considered were the *bf_interval* and the *bf_resolution* parameters. The *bf_interval* determines the interval between the backfill scheduling scans⁸. Initially, this parameter was changed from the default of 30 seconds to 1 second. This increase in backfill scanning frequency was expected to increase the number of backfillable jobs that would be detected on the *Slurm Training Cluster*, bringing it more into alignment with the SimPE model. In addition to this, the *bf_resolution* parameter was also reduced from 1 minute to 1 second. This parameter determines the resolution of the scheduling table that *Slurm* builds up in memory, based on the estimated start and end time of jobs. This scheduling table is used to determine whether a job can be backfilled or not. By increasing this resolution, it was expected that

⁸https://slurm.schedmd.com/sched_config.html

even more backfillable jobs would be detected, since smaller gaps in the scheduling table would now be visible to the scheduler. As was described in Section 4.4, the SimPE model also used a resolution of 1 second for determining whether a job can be backfilled.

Unfortunately, even though updating both of these parameters did work as described, there was another unforeseen problem. *Slurm* requires a job to submit a wall time or maximum run time, which is used to create the scheduling table. However, the wall time is always rounded up to the next minute⁹. This effectively created a minimum run time for jobs that could be backfilled, and made it hard to test the SimPE model without adding more features to the model. Alternatively, longer average job run times could have been used, but as previously described this was not possible for this project. Due to these complications, we opted to compare and analyse the differences between all three models for this project. The only Slurm configuration option that was changed was the scheduling type, which changed from the basic scheduler to the backfill scheduler (i.e., *Scheduler-Type=sched/backfill*). All other configuration options were left the same as for the FIFO simulation run.

5.4.6 Multi-resource

The previous simulation runs focused on the testing of different scheduling types (e.g., backfill), but the jobs only made use of a single resource type (i.e., CPUs). This section will instead focus on validating the effect of jobs using more than one resource type (i.e., CPU and RAM). This is particularly relevant for this project, since the SimPE model only models the multiple resources requested by a computing jobs using a single PE unit measurement. In order to test the effect of the PE-based modelling for multi-resource jobs, two test simulation runs were created. This project focused only on the simplest FIFO scheduling type and used 500 jobs per job batch. The job types used in the FIFO simulation run in Section 5.4.1 was used as a base (i.e., 1 PE and 3 PE jobs), with each job requesting the same amount of CPU and RAM resources (in PE units). For the first multi-resource simulation run, a single 3 PE job was included where that requested more RAM (i.e., 3 PE) than CPUs (i.e. 1 PE). Table 5.4 shows the details of these three job types of this simulation run. Notably, the

⁹<https://slurm.schedmd.com/sbatch.html>

two 3 PE jobs had a frequency of 25% each, while the 1 PE job had a frequency of 50%. An equal 50-50 split was therefore maintained for these testing runs. This allowed the same job inter-arrival rate to be used as for the original FIFO simulation runs. The system utilisation is therefore expected to still be close to 60%, with a slight reduction due to the newly added jobs requesting less CPUs.

Table 5.4: Job types used in Multi-resource runs (three job types version).

Job PE	CPUs	RAM (PE)	RAM (GB)	Mean Runtime (sec)	Frequency
1	1	1	7.42	30	50%
3	1	3	22.27	30	25%
3	3	3	22.27	30	25%

Although this simulation run did have jobs requesting different amounts of CPU and RAM resources, the two resources were not equally important. This can be seen by the fact that the Job PE value is always equal to the RAM PE value of each job.

For the second multi-resource simulation run, both resources types were made equally important by adding another 3 PE job type. Specifically, a 3 PE job was added where only 1 PE worth of RAM was requested, while 3 CPUs were requested. The resulting four job types are shown in Table 5.5.

Table 5.5: Job types used in Multi-resource runs (four job types version).

Job PE	CPUs	RAM (PE)	RAM (GB)	Mean Runtime (sec)	Frequency
1	1	1	7.42	30	50%
3	1	3	22.27	30	16.67%
3	3	1	7.42	30	16.67%
3	3	3	22.27	30	16.67%

Similar to the previous simulation run, the 50-50 split was maintained between 1 PE and 3 PE jobs, leading the three 3PE jobs to be split into 16.67% portions.

Lastly, in terms of Slurm configuration the same configuration files were used as was for the FIFO simulation runs.

5.5 Selected Job Configurations

The job configurations used for test simulation runs described in this chapter were carefully chosen to balance several competing requirements. For

basic FIFO scheduling behaviour, the combination of 1 PE and 3 PE jobs was good enough to demonstrate some core scheduling functionality while maintaining reasonable run times. When testing priority scheduling, we used the same job sizes but introduced priority levels, allowing us to observe the effect of both size and priority without having overly complex interactions. For backfill testing, the addition of a very small number of 8 PE jobs created the required queue dynamics to test backfill behaviour, while a difference in job durations (30 seconds for 1 PE jobs versus 60 seconds for 3 PE jobs) increased the opportunities for backfilling to occur. Finally, for multi-resource testing, we designed job types that would specifically highlight any differences between PE-based modelling and actual multi-resource allocation, using combinations of CPU and RAM requests that would challenge our simplified PE model. In the next chapter, we analyse the results of using these job configurations for simulation runs and test how well the SimPE model was able to capture the behaviour of these carefully selected job scenarios.

Chapter 6

Results

In this chapter, we will discuss the results from the test simulation runs that compare the SimPE model to the same runs on the *Slurm Training Cluster*. As discussed in Chapter 5, ten independent simulation runs were performed for each scheduling type. This included the FIFO, Static Priority, Backfill, and Dynamic Priority scheduling types. Results obtained from the simulation runs that investigated the impact of jobs requesting multiple resources (e.g. CPU and RAM) are also discussed.

6.1 FIFO

The FIFO scheduling is the simplest scheduling type for which simulation runs were performed. A summary of the results for the ten runs are presented in Table 6.1, with the average waiting time for the SimPE model indicated as $\overline{W}_{PE}^{(fifo)}$ and the *Slurm Training Cluster* as $\overline{W}_{ST}^{(fifo)}$. The average ratio between the waiting times of the two models (i.e., $\overline{W}_{PE}^{(fifo)} / \overline{W}_{ST}^{(fifo)}$) was 0.88, with a standard deviation of 0.05. Based on the minimum ratio of 0.8 and maximum ratio of 0.95, the *Slurm Training Cluster* waiting time was higher than the SimPE model waiting time for all ten runs. On average the *Slurm Training Cluster* waiting time was higher by 0.74 (7.28 – 6.54) seconds. It is therefore likely that there were processes increasing the waiting time of the *Slurm Training Cluster*, which weren't modelled in the SimPE model.

In order to investigate these differences, a visualisation is performed in Figure 6.1 of the first FIFO simulation run (i.e., simulation run 1 of

Table 6.1: Comparison of the mean job waiting time (in seconds) for FIFO scheduling between the SimPE model ($\overline{W}_{PE}^{(fifo)}$) and the Slurm Training Cluster ($\overline{W}_{ST}^{(fifo)}$) using multiple metrics (i.e., minimum, maximum, standard deviation and average) over the ten independent job batches. Each job batch had 200 jobs.

Metric	$\overline{W}_{PE}^{(fifo)}$	$\overline{W}_{ST}^{(fifo)}$	$\overline{W}_{PE}^{(fifo)} / \overline{W}_{ST}^{(fifo)}$
min	3.13	3.93	0.80
max	13.43	14.21	0.95
s_{dev}	3.17	3.17	0.05
average	6.54	7.28	0.88

10), displaying the waiting time for both the *Slurm Training Cluster* and SimPE model on an individual job level. The straight orange line represents SimPE’s waiting time, while the dotted blue line indicates the *Slurm Train Cluster* waiting time. Overall, both models’ waiting times follow a similar trend, with the periods of high waiting times having very similar values. However, one notable difference is the greater variability or jitter observed in the *Slurm Training Cluster* waiting times. The jitter appears to be predominantly in one direction (i.e, increasing the waiting time of the *Slurm Training Cluster*). This is particularly evident during periods when the SimPE’s waiting times remain constant at 0 seconds, while the *Slurm Training Cluster* waiting times have jitter of 1 to 2 seconds.

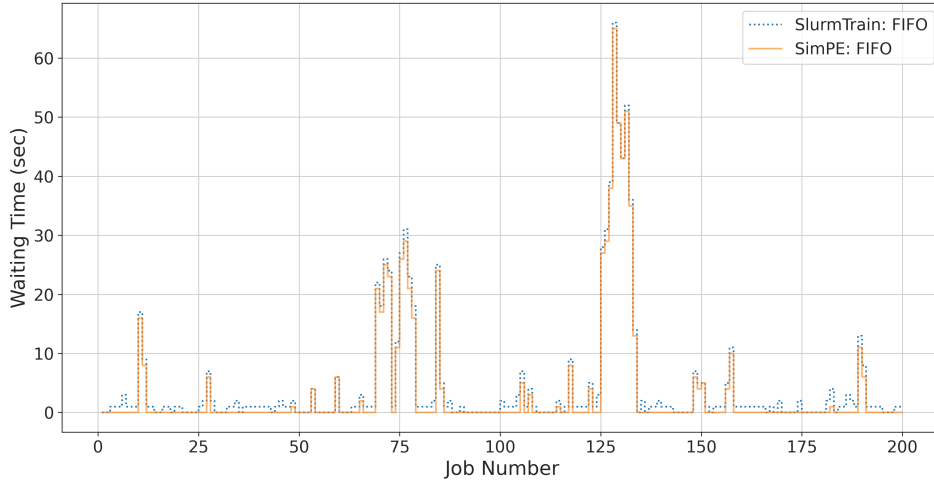


Figure 6.1: A plot of the job waiting times for the first FIFO simulation run, with the Slurm Training Cluster times indicated by a dotted blue line and SimPE times indicated by a solid orange line.

Lastly, we also summarise the model waiting times by job type in Table 6.2. The same ten job batches were used, but the average waiting time was calculated for each job type in a job batch. Subsequently, the average was taken over the ten job batches, denoted as $\mu_{\overline{W}_{PE}^{(fifo)}}$ for the SimPE

model and $\mu_{\overline{W}_{ST}^{(fifo)}}$ for the *Slurm Training Cluster*. The table highlights the disparity in waiting times between the 1 PE and 3 PE jobs. In the case of the *Slurm Training Cluster*, the 1 PE jobs had an average waiting time of 4.84 seconds, which was roughly half the waiting time of 3 PE jobs of 9.65 seconds. A similar effect was seen for the SimPE model, with 1 PE jobs having an average waiting time of 4.13 seconds and 3 PE jobs having an average waiting time of 8.9 seconds. This effect is expected, since the 3 PE jobs request more resources, which often take longer to become available on the cluster.

Table 6.2: Comparison of the mean job waiting time (in seconds) by job type between the *Slurm Training Cluster* and *SimPE* model using FIFO scheduling for ten independent job batches.

Job Type	Average Jobs Per Batch	$\mu_{\overline{W}_{PE}^{(fifo)}}$	$\mu_{\overline{W}_{ST}^{(fifo)}}$	$\mu_{\overline{W}_{PE}^{(fifo)}} / \mu_{\overline{W}_{ST}^{(fifo)}}$
1 PE	99.40	4.13	4.84	0.85
3 PE	100.60	8.90	9.65	0.92

6.2 Priority

This section will show the results for both the static priority and dynamic priority runs. As was described in Section 5.4.2, the static priority simulation runs kept job priorities stable throughout the simulation run. The dynamic priority runs on the other hand increased job priorities, for every minute spent in the pending queue of jobs.

6.2.1 Static Priority

A summary of the static priority results is shown in Table 6.3, indicating the average waiting time on the SimPE model ($\overline{W}_{PE}^{(pri)}$) and the *Slurm Training Cluster* ($\overline{W}_{ST}^{(pri)}$). The ratio between the waiting times of the two models (i.e, $\overline{W}_{PE}^{(pri)} / \overline{W}_{ST}^{(pri)}$) can be seen to range between 0.82 and 1.08. Since the maximum ratio is higher than one, it means that unlike the FIFO scheduling runs, there were some average waiting times where $\overline{W}_{PE}^{(pri)} > \overline{W}_{ST}^{(pri)}$. In fact out of the ten runs, there were two runs where this was the case. It is also clear that the range of average waiting times was larger than for the FIFO scheduling runs. The FIFO Scheduling waiting times for the *Slurm Training Cluster* average waiting time ranged from 3.93 to 14.21 seconds, with only one value above 10.0 seconds. However,

for priority scheduling the *Slurm Training Cluster* average waiting time ranged from 4.43 to 29.06 seconds, and had four values above 20 seconds.

Table 6.3: Comparison of the mean job waiting time (in seconds) for Priority scheduling between the SimPE model ($\overline{W}_{PE}^{(pri)}$) and the *Slurm Training Cluster* ($\overline{W}_{ST}^{(pri)}$) using multiple metrics (i.e., minimum, maximum, standard deviation and average) over the ten independent job batches. Each job batch had 500 jobs.

Metric	$\overline{W}_{PE}^{(pri)}$	$\overline{W}_{ST}^{(pri)}$	$\overline{W}_{PE}^{(pri)} / \overline{W}_{ST}^{(pri)}$
min	3.67	4.43	0.82
max	31.51	29.06	1.08
s_{dev}	9.43	8.87	0.08
average	14.39	15.07	0.92

A summary of the results by job type is shown in Table 6.4, with the batch mean of the average waiting time denoted by $\mu_{\overline{W}_{PE}^{(pri)}}$ for the SimPE model and $\mu_{\overline{W}_{ST}^{(pri)}}$ for the *Slurm Training Cluster*. Similar to the FIFO results, the 3 PE jobs had higher waiting times than the 1 PE jobs.

Table 6.4: Comparison of the mean job waiting time (in seconds) by job type between the *Slurm Training Cluster* and SimPE model using Priority scheduling for ten independent job batches.

Job Type	Average Jobs Per Batch	$\mu_{\overline{W}_{PE}^{(pri)}}$	$\mu_{\overline{W}_{ST}^{(pri)}}$	$\mu_{\overline{W}_{PE}^{(pri)}} / \mu_{\overline{W}_{ST}^{(pri)}}$
1 PE - Low Priority	123.90	15.13	19.46	0.78
3 PE - Low Priority	124.00	22.41	26.96	0.83
1 PE - High Priority	122.20	6.70	3.95	1.70
3 PE - High Priority	129.90	12.99	9.72	1.34

The 1 PE high priority job had a waiting time of 3.95 seconds, while the 3 PE high priority job had a waiting time of 9.72 seconds. Both models followed this pattern. As would be expected, all lower priority jobs also had higher waiting times than the high priority jobs. For example, the *Slurm Training Cluster* waiting time for 1 PE low priority jobs was 19.46 seconds, compared to the high priority waiting time of 9.72 seconds. This was also the case for both models. Lastly, the ratios between models (i.e. $\mu_{\overline{W}_{PE}^{(pri)}} / \mu_{\overline{W}_{ST}^{(pri)}}$), for low priority jobs were lower than 1.00 (i.e, 0.78 and 0.83), whereas the ratios between models for high priority jobs were higher than 1.00 (i.e., 1.70 and 1.34). This is more unusual, since in the FIFO Scheduling results, the SimPE waiting times were always lower than the *Slurm Training Cluster* waiting times. The reason for seeing these lower waiting times is unclear. It is possible that both were caused by the jitter observed on the *Slurm Training Cluster*, but that the cumulative jitter of all the jobs affected lower and higher priority jobs waiting times in different ways. Alternatively, the difference could be a subtle modelling difference

the priority scheduling between the SimPE model and *SlurmTrain*. Unfortunately, without isolating the effect of the jitter on the waiting times, it is hard to say anything conclusive.

Similarly to the FIFO results, we also investigated the waiting time of individual jobs within one of the simulation runs. A visualisation of the waiting times of each individual job of the first Priority simulation run is shown in Figure 6.2, showing both the *Slurm Training Cluster* and SimPE model waiting times. Similarly to the results in section 6.1, there

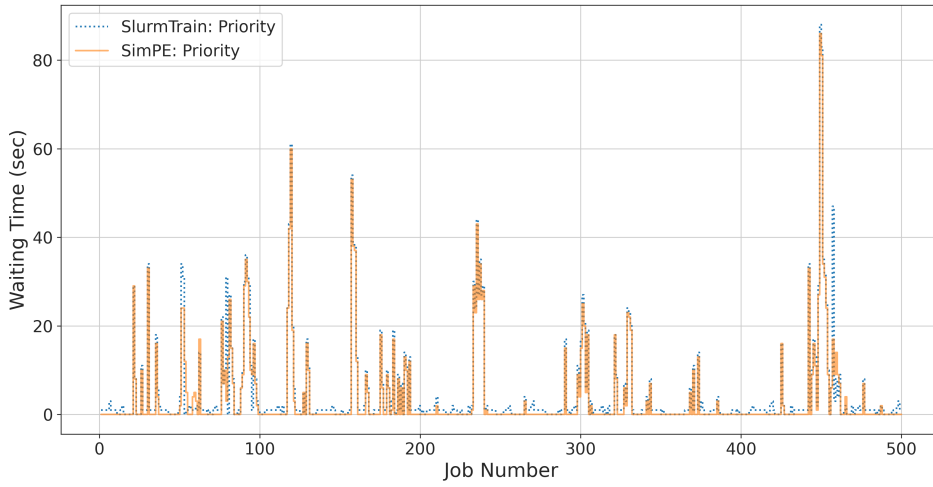


Figure 6.2: A plot of the job waiting times in Priority run number 1, with the *Slurm Training Cluster* times indicated by a dotted blue line and SimPE times indicated by a solid orange line.

was also a level of jitter observed for *Slurm Training Cluster* for waiting times close to zero. Focusing on the time periods where the waiting times were not close to zero, it appears that the general level of the waiting times was higher than for the first FIFO scheduling run. In fact, most of the high waiting time periods exceed 20 seconds. For most of these high waiting time periods, the SimPE and *Slurm Training Cluster* can seem to match quite closely. However, there are a few cases where there are mismatches. For this one simulation run, the mismatches appear to be a few seconds difference in waiting times, with the *Slurm Training Cluster* model generally having a higher waiting time. However, towards the end of the simulation run there is one case where there is a larger waiting time spike for both models, with the *Slurm Training Cluster* having only a slightly higher waiting time. However, directly afterwards the *Slurm Training Cluster* has a second waiting time spike that is much higher than that observed for the SimPE model.

6.2.2 Dynamic Priority

The results for the dynamic priority runs, broken down by job type are shown in Table 6.5. As was discussed in section 5.4.4, the main difference when compared to static priority runs is that the priority values of jobs were increased for every minute spent waiting in the queue. The priority values shown in the table for each job type (i.e., 5,000 and 10,000) were therefore only the starting priority values of the jobs. However, this added complexity could make the results harder to interpret. In order to make interpretation easier, the same ten job batches that were used for the static priority runs, were also used for the dynamic priority runs. In the last column of the table, the ratio between the SimPE model and *Slurm Training Cluster* for the dynamic priority runs ($\mu_{\overline{W}_{PE}}^{(pri2)} / \mu_{\overline{W}_{ST}}^{(pri2)}$) is shown to range between 0.84 and 1.07. The dynamic priority model therefore had closer results than the static priority runs, which had a much wider range of 0.78 to 1.7.

Table 6.5: Comparison of the mean job waiting time (in seconds) by job type between the SimPE model and Slurm Training Cluster using Dynamic Priority scheduling over ten independent job batches. The Job Type indicates each job's size and priority. The waiting times are compared between the two models, but also between the dynamic priority (pri2) and static priority (pri) simulation runs.

Job Type	Average Jobs Per Batch	$\mu_{\overline{W}_{PE}}^{(pri2)}$	$\mu_{\overline{W}_{ST}}^{(pri2)}$	$\mu_{\overline{W}_{PE}}^{(pri2)} / \mu_{\overline{W}_{PE}}^{(pri)}$	$\mu_{\overline{W}_{ST}}^{(pri2)} / \mu_{\overline{W}_{ST}}^{(pri)}$	$\mu_{\overline{W}_{PE}}^{(pri2)} / \mu_{\overline{W}_{ST}}^{(pri2)}$
1 PE - 5,000 prio	123.90	14.70	17.56	0.97	0.90	0.84
3 PE - 5,000 prio	124.00	20.84	23.40	0.93	0.87	0.89
1 PE - 10,000 prio	122.20	7.89	7.34	1.18	1.86	1.07
3 PE - 10,000 prio	129.90	14.23	13.39	1.10	1.38	1.06

In order to better understand the reason for these differences, we compare the model waiting times between dynamic priority and static priority runs. Starting with the SimPE low priority jobs, the ratio between the dynamic priority and static priority model ($\mu_{\overline{W}_{PE}}^{(pri2)} / \mu_{\overline{W}_{PE}}^{(pri)}$) was equal to 0.97 for 1 PE jobs and 0.93 for 3 PE jobs. Both indicate that the dynamic priority waiting times were lower than the static priority waiting times. In the case of the *Slurm Training Cluster* low priority jobs, the decrease was in the same direction, with a ratio ($\mu_{\overline{W}_{ST}}^{(pri2)} / \mu_{\overline{W}_{ST}}^{(pri)}$) of 0.9 for 1 PE jobs and 0.87 for 3 PE jobs. This decrease in low priority waiting times was expected. Since the priority values of queuing jobs keep increasing, this means that a low priority job (i.e. starting with a priority value of 5,000) that has been queuing for long enough could increase its priority to be higher 10,000 or higher. This means that it would take priority over any incoming high

priority jobs and would therefore have lower waiting time.

For high priority jobs, the SimPE waiting times increased for the dynamic priority model, with $\mu_{\overline{W}_{PE}^{(pri2)}} / \mu_{\overline{W}_{PE}^{(pri)}}$ having a value of 1.18 for 1 PE jobs and 1.10 for 3 PE jobs. An increase in waiting time for these jobs was also expected, since an incoming high priority job with a priority of 10,000 is not necessarily the highest priority job in the queue any more. In the case of the *Slurm Training Cluster* runs, the increase in waiting times of high priority jobs was even larger, with the waiting times of 1 PE jobs increasing with a ratio of 1.86 and 3 PE jobs with a ratio of 1.38. Taking the 1 PE high priority job as an example, the *Slurm Training Cluster* average waiting time increased from 3.95 seconds for the static priority run (Table 6.4) to 7.34 seconds for the dynamic priority run. These increases brought the waiting times for high priority jobs back into line with the average waiting time of the SimPE Model (e.g. 7.89 seconds), and likewise for the 3 PE jobs. The *Slurm Training Cluster* waiting times were still slightly higher though.

6.3 Backfill

The results for the backfill validation runs are shown in Table 6.6, including the average waiting times from the ten backfill runs for the three models, with the Slurm Simulator (SS) results indicated using $\overline{W}_{SS}^{(bf)}$. The general level of these waiting times was higher than those observed in the FIFO and priority validation runs. For example, the highest *Slurm Training Cluster* waiting time is 284.57 seconds, compared to the 29.06 seconds observed in the priority validation run (shown in Table 6.3). This increase in waiting time is just a reflection of the change in the job types, specifically, the addition of the 8 PE jobs, but also the increased run duration of 3 PE jobs. The table also includes the ratios of the waiting times relative to the *Slurm Training Cluster* waiting times. Notably, the SimPE model waiting times were all significantly lower than the *Slurm Training Cluster* waiting times, with ratios ($\overline{W}_{PE}^{(bf)} / \overline{W}_{ST}^{(bf)}$) ranging from 0.46 to 0.81. The *Slurm Simulator* ratios ($\overline{W}_{PE}^{(bf)} / \overline{W}_{ST}^{(bf)}$) on the other hand were much closer and ranged from 0.91 to 1.1. In order to gain a better understanding of the driver of the lower SimPE model waiting times, the results were also analysed at a job type level.

Table 6.6: Comparison of the mean job waiting time (in seconds) for Backfill scheduling between the SimPE model ($\overline{W}_{PE}^{(bf)}$), Slurm Simulator model ($\overline{W}_{SS}^{(bf)}$) and the Slurm Training Cluster ($\overline{W}_{ST}^{(bf)}$) using multiple metrics (i.e., minimum, maximum, standard deviation and average) over the ten independent job batches. Each job batch had 200 jobs.

Metric	$\overline{W}_{PE}^{(bf)}$	$\overline{W}_{SS}^{(bf)}$	$\overline{W}_{ST}^{(bf)}$	$\overline{W}_{PE}^{(bf)} / \overline{W}_{ST}^{(bf)}$	$\overline{W}_{SS}^{(bf)} / \overline{W}_{ST}^{(bf)}$
min	12.62	16.81	17.43	0.33	0.91
max	130.27	258.77	284.57	0.81	1.10
s_{dev}	35.56	72.12	77.79	0.15	0.07
average	47.39	87.45	87.01	0.60	1.01

The waiting times are also shown broken down by job type in Table 6.7 for each of the three models. The 1 PE jobs had a ratio of 0.14 between SimPE model and *Slurm Training Cluster* waiting times. The 1 PE job waiting times were therefore significantly lower than the *Slurm Training Cluster*. The 3 PE job waiting times were also lower, but closer to a SimPE to *Slurm Training Cluster* ratio of 0.79. The main cause for the lower waiting times is that the SimPE model is backfilling much more aggressively than the *Slurm Training Cluster*. The main driver of this is the fact that the SimPE model does not have waiting periods or intervals between backfill scheduling scans. *Slurm* on the other hand has a default interval of 30 seconds between backfill scheduling scans¹. One observation that seems to contradict this explanation, is the ratio of 2.74 observed for the SimPE model when compared to the *Slurm Training Cluster* for the 8 PE job. As previously explained in Section 4.3.4, backfilling on *Slurm* does not delay the start of a higher priority job. However, in this batch all jobs were of the same priority, making it possible for a backfilled job to delay the start of another job. This is likely what is happening with the 8 PE Jobs, and due to the aggressive backfilling the size of this effect is larger than for the other models.

Table 6.7: Comparison of the mean job waiting time (in seconds) by job type between the SimPE model, Slurm Simulator model and the Slurm Training Cluster, all using Backfill scheduling for ten independent job batches.

Job Type	Average Jobs Per Batch	$\mu_{\overline{W}_{PE}}^{(bf)}$	$\mu_{\overline{W}_{SS}}^{(bf)}$	$\mu_{\overline{W}_{ST}}^{(bf)}$	$\mu_{\overline{W}_{PE}}^{(bf)} / \mu_{\overline{W}_{ST}}^{(bf)}$	$\mu_{\overline{W}_{SS}}^{(bf)} / \mu_{\overline{W}_{ST}}^{(bf)}$
1 PE	100.60	10.01	74.70	72.82	0.14	1.03
3 PE	97.60	79.11	99.67	100.43	0.79	0.99
8 PE	1.80	448.19	163.65	163.33	2.74	1.00

This effect can be seen more clearly in Figure 6.3, which is a plot that compares the waiting time of each individual job between the *Slurm Training Cluster* and SimPE for Backfill run number 1. The initial high waiting

¹<https://slurm.schedmd.com/slurm.conf.html>

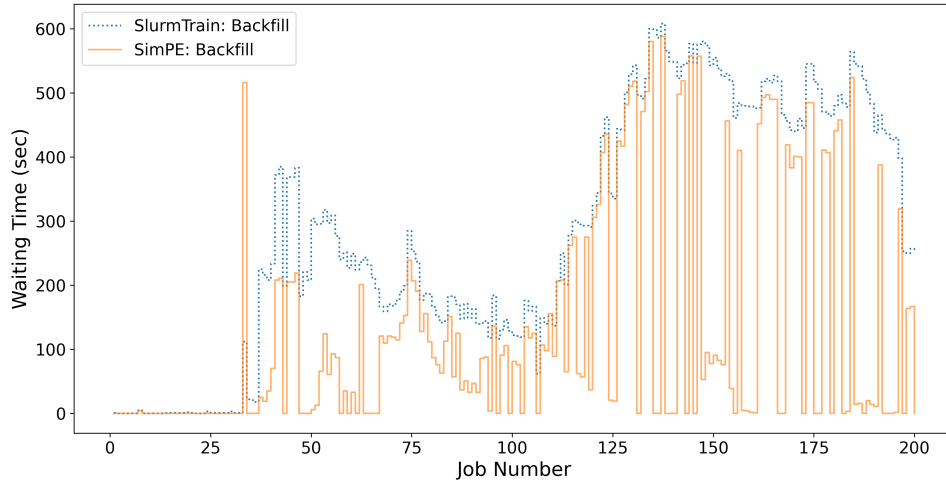


Figure 6.3: A plot of the job waiting times in Backfill run number 1, with the *Slurm Training Cluster* times indicated by a dotted blue line and *SimPE* times indicated by a solid orange line.

time for the *SimPE* model of more than 500 seconds is for an 8 PE job waiting time. The waiting time for same job on the *Slurm Training Cluster* was approximately 100 seconds. Since on average there were fewer than two 8 PE Jobs per job batch, it also makes sense that these waiting time were close to the average waiting times for 8 PE jobs in Table 6.7 (i.e., 448.18 seconds for *SimPE* and 100.43 seconds for *SlurmTrain*). After the initial 8 PE job, there were other large differences between the models. Specifically, the *SimPE* waiting time often dropped to a low waiting time, and thereafter increased to waiting time that was more similar to the *Slurm Training Cluster* waiting time. The jump to a similar waiting time was caused by 3 PE jobs. The reduction to 0 second waiting times was due to 1 PE jobs that got backfilled with very low or even zero waiting times.

The individual jobs' waiting times were also visualised for the *Slurm Simulator* model, together with the *Slurm Training Cluster* in Figure 6.4. The *Slurm Simulator* model follows the *Slurm Training Cluster* quite closely. However, there is some variability around the *Slurm Training Cluster* waiting times. The high level of waiting times observed and the general upward trend for waiting times, which could indicate that the cluster was not in a stable state for much of this run. For comparison purposes, one of the other backfill simulation runs was also visualised and they had more stable behaviour, with the exception of a high waiting times spike for both models due to the 8 PE jobs.

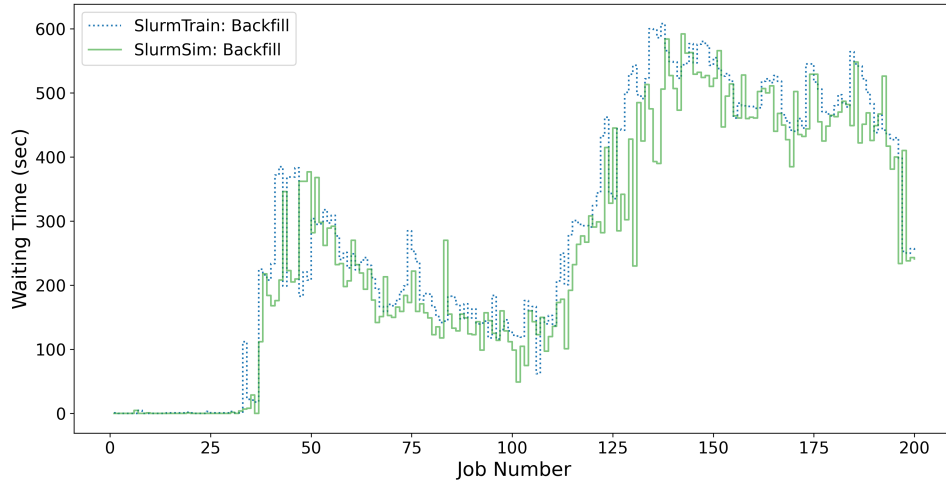


Figure 6.4: A plot of the job waiting times in Backfill run number 1, with the *Slurm Training Cluster* times indicated by the dotted blue line and *Slurm Simulator* times indicated by a solid green line.

6.4 Multi-Resource

This section presents the results of the simulation runs where job types used multiple resource types (e.g. CPU and RAM) for each computing job. For the previous simulation runs (i.e. FIFO, Priority and Backfill), only a single resource type was used (i.e. CPU), since the main focus was the testing of the scheduling type. The focus of this section is to test the effect that multiple resource jobs have on the SimPE model and to compare against the *Slurm Training Cluster* results. As was discussed in Section 5.4.6, only the FIFO scheduling type was considered for these simulation runs. Two different scenarios were run, one using three job types and four job types, for which the results will be shown below.

6.4.1 Three Job Types

The results for the three job type multi-resource run are shown in Table 6.8. The ratio between the SimPE model and the *Slurm Training Cluster* results ranged from 0.9 to 0.95, with an average ratio value of 0.93 and standard deviation for the ratio value of 0.01. For comparison, the FIFO simulation runs in Section 6.1 had an average ratio value of 0.88, with standard deviation of 0.05. The three job type multi-resource runs therefore seem to have lower variability. However, it should be taken into account that multi-resource batches had 500 jobs per batch, while the

FIFO simulation run had only had 200 jobs per batch.

Table 6.8: Comparison of the mean job waiting time (in seconds) for Multi-resource runs (with three job types), between the SimPE model ($\overline{W}_{PE}^{(mr3)}$) and the Slurm Training Cluster ($\overline{W}_{ST}^{(mr3)}$) using multiple metrics (i.e., minimum, maximum, standard deviation and average) over the ten independent job batches. Each job batch had 500 jobs.

Metric	$\overline{W}_{PE}^{(mr3)}$	$\overline{W}_{ST}^{(mr3)}$	$\overline{W}_{PE}^{(mr3)} / \overline{W}_{ST}^{(mr3)}$
min	7.28	8.11	0.90
max	14.99	15.85	0.95
s_{dev}	2.43	2.46	0.01
average	10.81	11.61	0.93

A breakdown of the waiting times by job type is shown in Table 6.9. However, one difference to the multi-resource job types is that each job types name starts with the overall PE value (e.g. 3 PE - 1 CPU(s), 3 RAM (in PE)). In addition, the underlying resource values are also shown, but measured in PE units, to make it easier to compare different resources values. The ratio ($\mu_{\overline{W}_{PE}^{(mr3)}} / (\mu_{\overline{W}_{ST}^{(mr3)}})$) were similar across the different job types, only ranging from 0.91 to 0.94. As expected, the 1 PE job had the lowest waiting time, while the 3 PE jobs had higher waiting times. Between the two 3 PE jobs, the “1 CPU and 3 RAM (in PE)” job type had a slightly lower waiting time. However, these differences could just be due to randomness, especially for the SimPE model, which would have modelled these two job types both as identical 3 PE jobs.

Table 6.9: Comparison of the mean job waiting time (in seconds) by job type between the Slurm Training Cluster and SimPE model for Multi-resource runs using 3 different job types, over ten independent job batches.

Job Type	Average Jobs Per Batch	$\mu_{\overline{W}_{PE}^{(mr3)}}$	$\mu_{\overline{W}_{ST}^{(mr3)}}$	$\mu_{\overline{W}_{PE}^{(mr3)}} / \mu_{\overline{W}_{ST}^{(mr3)}}$
1 PE - 1 CPU(s), 1 RAM (in PE)	253.10	8.31	9.10	0.91
3 PE - 1 CPU(s), 3 RAM (in PE)	124.80	13.24	14.08	0.94
3 PE - 3 CPU(s), 3 RAM (in PE)	122.10	13.43	14.24	0.94

In order to highlight the impact of multi-resource jobs on the SimPE model, a PE-time plot was used. The first 25 minutes of the first simulation run are visualised in Figure 6.5, showing the PE available over time for both the SimPE model and the *Slurm Training Cluster*. In this case the two models were almost identical, however as will be seen in the four jobs section, it is an important reference figure to understand the effect of multi-resource jobs on the SimPE model.

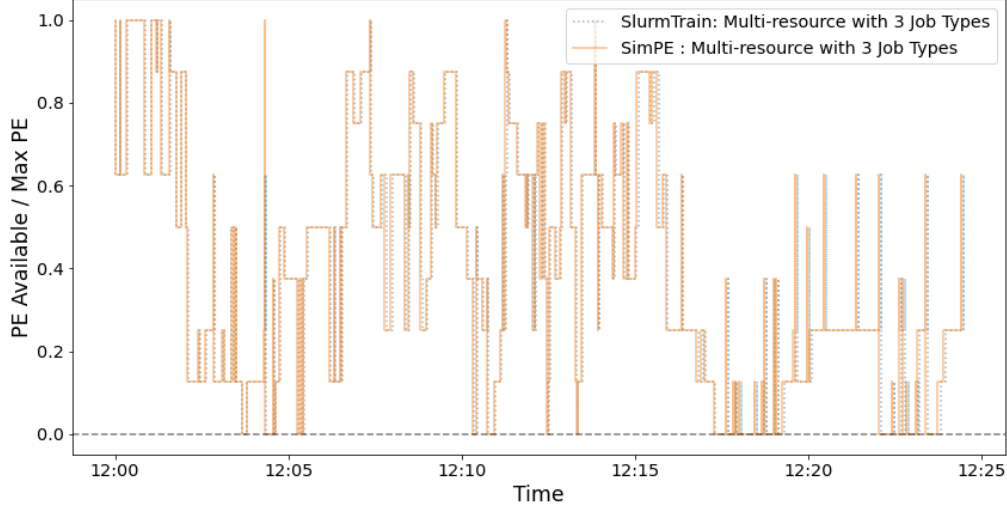


Figure 6.5: A PE Time Plot of the first 25 minutes of the Multi-resource simulation run with 3 Job types between the Slurm Training Cluster shown in blue, and the SimPE model shown in orange.

6.4.2 Four Job Types

The results of the four job type multi-resource simulation runs are shown in Table 6.10. The range of the ratio between the models ($\overline{W}_{PE}^{(mr4)} / \overline{W}_{ST}^{(mr4)}$) was between 1.08 and 1.83, with an average value of 1.39. This is much higher than the average ratio of 0.93 for the three job type runs in Section 6.4.1. In fact, the three job type simulation runs ratios were always lower than 1.00, meaning the SimPE waiting times were always lower. For these runs, the opposite was observed, with the ratios always being higher than 1.00, meaning the SimPE waiting times were always higher. This increase in waiting times is mainly due to the limitations of the SimPE model, which can only model the highest resource used for each job type. In other words, SimPE model always makes the most conservative assumption between the CPU and RAM requested.

Table 6.10: Comparison of the mean job waiting time (in seconds) for Multi-resource runs (with four job types), between the SimPE model ($\overline{W}_{PE}^{(mr4)}$) and the Slurm Training Cluster ($\overline{W}_{ST}^{(mr4)}$) using multiple metrics (i.e., minimum, maximum, standard deviation and average) over the ten independent job batches. Each job batch had 500 jobs.

Metric	$\overline{W}_{PE}^{(mr4)}$	$\overline{W}_{ST}^{(mr4)}$	$\overline{W}_{PE}^{(mr4)} / \overline{W}_{ST}^{(mr4)}$
min	6.62	3.95	1.08
max	14.75	11.27	1.83
s_{dev}	2.45	2.22	0.22
average	9.78	7.21	1.39

A breakdown of the results by job type is shown in Table 6.11. The table is similar to the three job type table in the previous section, but with the extra 3 PE job type that was added, namely "3 CPU and 1 RAM (in PE)". Looking at the ratios of the models by job type, it is clear that the SimPE

Table 6.11: Comparison of the mean job waiting time (in seconds) by job type between the Slurm Training Cluster and SimPE model for Multi-resource runs using four different job types, over ten independent job batches.

Job Type	Average Jobs Per Batch	$\mu_{\overline{W}_{PE}}^{(mr4)}$	$\mu_{\overline{W}_{ST}}^{(mr4)}$	$\mu_{\overline{W}_{PE}}^{(mr4)} / \mu_{\overline{W}_{ST}}^{(mr4)}$
1 PE - 1 CPU(s), 1 RAM (in PE)	258.00	7.38	5.61	1.31
3 PE - 1 CPU(s), 3 RAM (in PE)	83.00	12.56	8.70	1.44
3 PE - 3 CPU(s), 1 RAM (in PE)	77.40	11.53	7.80	1.48
3 PE - 3 CPU(s), 3 RAM (in PE)	81.60	12.87	10.08	1.28

Model had higher waiting times for all the job types, with ratios ranging from 1.28 to 1.48. Although, it could just be random variation, it is still interesting to note that the first job type (1 CPU, 1 PE RAM) and last job type (3 CPUs, 3 PE RAM) had lower ratios, namely 1.31 and 1.28. For these jobs the CPU and RAM resources matched, and the assigned PE value for each job would have matched its resources usage. The SimPE would therefore have modelled these jobs correctly. For the second job type (1 CPU, 3 PE RAM) and third job type (3 CPUs, 1 PE RAM), the ratios were higher, namely 1.44 and 1.48. For these job types, the assigned PE value would have been the maximum PE value between the two resource types. Therefore, in both cases a 3 PE value would have been used. For these jobs, the SimPE model would have modelled the jobs more conservatively, implicitly assuming that this amount of resources was used in both CPU and RAM. The higher waiting times associated with these jobs likely indicate this fact. While, the higher waiting times on the other two job types are likely a knock-on effect of the conservative modelling of these two job types. It should also be noted that the variability of the 3 PE job waiting estimates in these runs was expected to be slightly higher in comparison with the 3 job type runs. This is due to the reduced number of 'average jobs per batch' for each 3 PE job, due to splitting the approximately 250 jobs allocated to 3 PE jobs between three job types instead of only two job types.

The available PE over time for the first 25 minutes of the first simulation run is visualised in Figure 6.6. The same job batch was used as in Figure 6.5 (i.e., the equivalent plot, in the case of 3 job types) and the same time frame of the run is also shown. This plot visualises why the SimPE

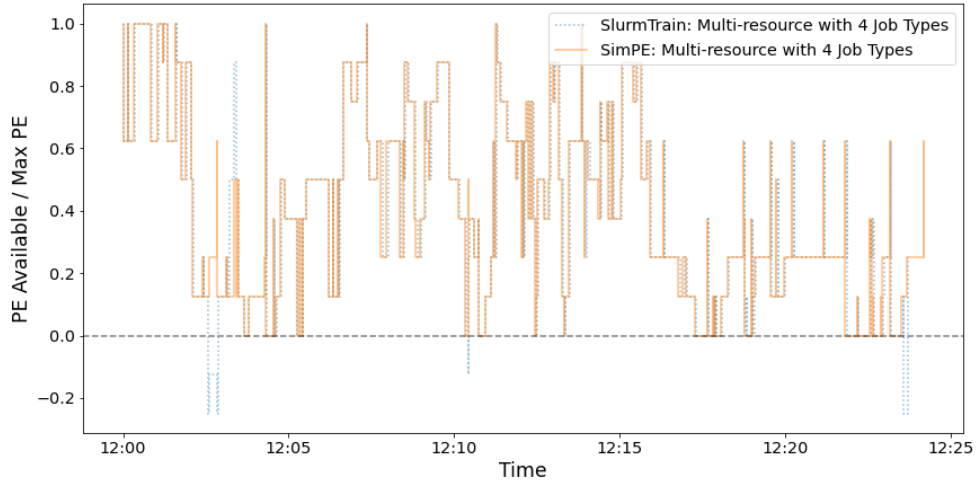


Figure 6.6: A PE Time Plot of the first 25 minutes of the Multi-resource simulation run with 4 job types between the Slurm Training Cluster shown in blue, and the SimPE model shown in orange.

model has more conservative waiting time estimates for the multi-resource jobs. Notably, the available PE for the SlurmTrain model goes negative in multiple times throughout the visualised period of the run. In reality, these jobs did run on the cluster so there were enough resources to execute them. However, the simplification made to assign a PE value to them can make the resources required for the jobs to run seem to be more onerous. As an example take the 1 CPU, 3 PE RAM job type, being assigned a 3 PE value. Even if there were 2 CPUs available on the cluster and 5 PE of RAM, this job would not run in the cluster, since the SimPE would model the cluster state as having only 2 PE available for executing jobs.

Chapter 7

Discussion and Conclusion

In this thesis, we investigated the feasibility of various tools to simulate a computing cluster and model the waiting time of jobs. We built the SimPE model, which is a discrete event simulation (DES) based model of the *Slurm* scheduler that models job resources using a single Processor Equivalent (PE) resource. The model focused on the scheduling policies used on the Ilifu cluster, specifically: FIFO, static and dynamic priority, as well as backfill scheduling policies. In order to assess the feasibility of the model, we set up a *Slurm Training Cluster* with a single small computing node to run simulations against the SimPE model. Additionally, we also set up a *Slurm simulator* model, which is a simulator based on the *Slurm* source code that is able to perform faster simulations of the *Slurm* scheduling policies. The *Slurm Simulator* used was found to be limited mainly to FIFO and backfill scheduling policies, but still proved useful for the development of the SimPE model. We then created multiple simulation types and assumptions to run on the different models and to assess the accuracy of the SimPE model for the different scheduling types. Most of the simulation types used two main job types, a low PE job type and a high PE job type, to make the results easier to interpret. We also assessed the SimPE model for jobs with multiple resource requests.

Before presenting the key findings from these assessments, it is useful to clarify the specific scope of the model and the experimental design. The model's main simplification is the use of a single Processor Equivalent (PE) unit to abstract away the complexities of multiple, distinct resources like CPU and RAM. While it specifically implements key scheduling policies like FIFO, static and dynamic priority, and backfill, other important features like the fairshare algorithm were deliberately excluded to maintain

focus. The model’s full architecture is detailed in Chapter 4, while the implications of these limitations are discussed in Section 7.2.

The experimental design itself followed a foundational principle for ensuring a fair comparison: whenever two or more systems were compared, they were always subjected to the exact same batch of jobs. While several factors were held constant (such as the hardware and job arrival process), the key variables that were systematically manipulated included the scheduling algorithm, specific job characteristics, and the priority calculation rules. A complete breakdown of the parameters for each testing run performed can be found in the tables presented in Chapter 5.

7.1 Key Findings

The results from running identical batches of jobs as testing runs on both the SimPE model and the *Slurm Training Cluster* (*SlurmTrain*) had the following key findings.

1. For the FIFO scheduling runs, the SimPE model closely matched *SlurmTrain* waiting times (i.e., a ratio of 0.88), but the jitter in job starting times on *SlurmTrain* caused its waiting times to be consistently higher.
2. For the Static Priority scheduling runs, the SimPE model closely matched the *SlurmTrain* waiting times on an overall level (i.e., ratio of 0.92). The *SlurmTrain* waiting times for *low* priority jobs were higher by a few seconds, also likely due to jitter. However, the waiting times of high priority jobs were *lower* by a few seconds.
3. The Dynamic Priority scheduling runs added the *Age* priority factor, and it had the expected effect on waiting times, reducing the advantage of high priority jobs. The results were even closer than for the Static Priority runs, especially for high priority jobs.
4. For the Backfill scheduling runs, the SimPE model had much lower waiting times (i.e., a ratio of 60%), mainly driven by its backfill implementation being more aggressive than *SlurmTrain*’s. The *Slurm Simulator* model, with its similar backfill implementation was also run for comparison purposes and had close results (i.e., ratio of 1.01).

5. For Multi-resource simulation runs, we tested the SimPE model’s ability to model jobs that make multiple resource requests. Using two resources, but with one resource still being the main bottleneck gave good results (i.e, ratio of 0.94). However, when both resources were equally important, the PE approach led to jobs being modelled conservatively, resulting in average waiting times of 9.78 seconds, compared to *SlurmTrain*’s 7.21 seconds (i.e, ratio of 1.39).

The model also fulfilled all the requirements that were initially set out in Section 4.1, suggesting that it would be useful as a scenario analysis tool, for example, in the case of analysing the waiting time jitter present in the *SlurmTrain* FIFO test. This finding also corroborates the main use case of Qsim Tang et al. (2009), where the DE-simulator tool’s main use is for scenario testing.

The model excels in situations where there is one main bottleneck in the computing cluster. In such scenarios, its simplicity and focus on Processor Equivalent (PE) resources provide clear insights into how different scheduling policies impact waiting times.

The model’s approach to estimating waiting times, especially in the context of jobs with multiple resource types, tends to be conservative. This conservative nature, while providing a safe estimate, may not always capture the nuanced and often unpredictable behaviour of complex multi-resource cluster environments. In real-world scenarios, the interplay between various types of resources can significantly impact job scheduling and waiting times, a factor that our current model may not fully account for.

7.2 Limitations of the Thesis

A limitation of the Dynamic Priority simulation run, is that the *Age* priority factor only affected jobs that were in the pending queue for longer than a minute. This is because it is the recalculation frequency of the Multi-priority plugin that was used for these runs, as described in Section 5.4.4. But as a result only a fraction of the jobs were affected by the *Age* factor. This also partly due to the low system utilisation target for simulation runs. When compared directly to the static priority runs – which were based on the same job batch files – the effect was large enough to be seen

in the waiting files of the different job types. However, it would be better if the dynamic priority scheduling could be tested in a way that influences more jobs.

On a more general note, the assumption was made that all job run times and job iter-arrival times follow a slightly modified exponential distribution. The details were discussed in Section 5.3.1. The main reason an exponential distribution was used is for simplicity, but is a limitation of the above simulation runs, since it limits their variability compared to bursty real-life jobs (Frachtenberg et al., 2005). Another reason for using a modified exponential distribution was that the SimPE model assumes that no two jobs would arrive at exactly the same time. However, if a future study requires more realistic job arrivals, this limitation could be removed with only minor changes to the model.

Another important assumption made for these simulation runs was that the job wall time was equal to a job’s actual run time. In reality, since *Slurm* would terminate a job running longer than its low wall time prematurely, users often use very conservative wall times (Soysal et al., 2018). *Slurm*’s backfill scheduler especially uses these estimates when determining which jobs to backfill¹. It is therefore a natural follow up study to do once the SimPE model’s backfill scheduler shows better results. Fortunately, it would take only minor changes to the SimPE model to enable job wall times to be used for such a future study.

An important limitation to point out about the *Slurm Training Cluster* was that it only had a single worker node with 8 CPUs and 64 GB of RAM. As a result, a limitation of these simulation runs was that the SimPE model was not tested against a cluster with multiple worker nodes. This will be left for a future study and will be discussed more in Section 7.3.

The fact that the worker node only had 8 CPUs did not limit the simulation runs significantly. It mainly meant that the SimPE model was only tested using a small number of concurrently running jobs (e.g. a maximum of eight 1 PE jobs).

A further methodological limitation is the reliance on descriptive statistics (i.e., the ratio of means) for comparing models, rather than on formal statistical hypothesis tests. This choice was necessitated by several factors.

¹https://slurm.schedmd.com/sched_config.html

The small number of independent simulation runs ($N = 10$) provided insufficient statistical power for robust testing. Furthermore, the waiting times of individual jobs within a single run are autocorrelated, violating the independence assumption crucial for tests like the Kolmogorov-Smirnov test. Finally, the presence of systemic 'jitter' in the physical cluster skewed the error distribution, complicating the application of tests that assume normality. A more sophisticated statistical evaluation that accounts for these factors remains a key opportunity for future work.

We assumed mainly two job types for the majority of the simulation runs (i.e, a 1 PE job and 3 PE job) and two priority levels. These were practical choices to simplify the simulation runs and make them easier to interpret. Even though the scheduling process was modelled as generally as possible, it does mean that the majority of the tests were done using these job types. In a future study it would therefore be prudent to do a simulation run using a more realistic job types, mainly as an additional check on the SimPE model.

Another limitation to point out is the small size of the job batch files used in the simulation runs, with job batch files ranging from 200 up to 500 jobs. These numbers were kept low because the *SlurmTrain* was run in real time. However, it does mean that there would be more variability in the resulting waiting time estimates for different job types. In order to compensate for this, every simulation run was performed independently 10 times. In addition, the system utilisation for each job batch file was tuned to decrease the probability of having an unstable system.

When calculating the average waiting times for each simulation or *SlurmTrain* run, the estimate was based on all jobs that were run. However, Bose (2013) recommends that warm-up (and cool-down) periods should be omitted from these periods, since the cluster is in an unstable state during those periods. As both models were run under identical conditions, the impact of including these periods on our comparative analysis should be minimal.

Although the tuning of the system utilisation of job batch runs seemed to be effective in generating more stable waiting times, it should be noted that the tuning was only applied on an overall level based on the mean arrival time and mean run time over a job batch file. However, as was shown in Section 3.1.2, ideally it should have been applied separately for

the low priority and high priority job types. This would have helped ensure that the low priority jobs did not become unstable in some of the simulation runs. However, the project did only use two priority classes and already targeted a conservative system utilisation of 60%, so this already reduced the risk of this occurring. The only cases where abnormally high system utilisation levels were observed were in the backfill simulation runs, where periods of high system utilisation were induced by the addition of one or two 8 PE jobs, specifically to test the backfill scheduling type. In particular, the first backfill simulation run that was performed and illustrated in Figure 6.3 showed signs of being in a unstable state, with a high waiting time even towards the end of the run. Comparing this run to the other backfill simulation runs showed this was the run with the highest mean waiting time at 284.57 seconds on the *Slurm Training Cluster*. The other nine backfill simulation runs, had lower waiting times and were not trending upwards and was therefore probably more stable.

Another limitation of the simulation runs performed is the fact that the average waiting times calculated over a whole job batch run were used when comparing models. However, in reality when making a waiting time estimate on an actual computing cluster, any waiting time estimate would have to be made in a forward looking fashion. A future study could therefore investigate running the job batch file on both models, but use the SimPE model to estimate the waiting time of pending jobs in a forward looking fashion throughout the simulation. This would make it possible to compare the SimPE model against *Slurm*'s built in job start time estimator in a similar way to N. Brown et al. (2022).

7.3 Future Studies

This research represents a step towards understanding the impact of scheduling policies on waiting times in the Ilifu cluster. By developing the SimPE model and conducting various tests, this thesis contributes to the nuanced understanding of how different scheduling approaches can affect cluster performance. Although the study may not revolutionise the field, it provides valuable insights and lays a foundation for more in-depth and broader research in computing cluster simulation.

In response to the limitations described in the previous section, we suggest

the following studies:

1. **Comprehensive Testing and Expansion:** Future research should focus on validating the SimPE model against clusters with multiple worker nodes, including a broader range of job types and resource combinations. This expansion is important for assessing the model's accuracy and scalability in more diverse and complex computing scenarios.
2. **Improvement of Supported Scheduling Algorithms:** Improving key scheduling algorithms within the SimPE model, particularly the backfill scheduler, is recommended. Additionally, developing and integrating an approximate fairshare scheduler would allow the model to simulate a wider array of scheduling policies. It would be especially beneficial when analysing the Ilifu cluster, since the fairshare has the heaviest weight on the cluster. These enhancements will increase the model's applicability and accuracy, especially in environments where different scheduling strategies are used.
3. **Incorporating and Validating the Impact of User Wall Times:** Improving the model to use user wall time estimates of different levels of conservativeness and validating the impact on different scheduling types could be useful.
4. **Estimation of Forward looking Wait Time Estimates:** The current model does not create forward looking wait time estimates. Adding this ability would make it possible to estimate the waiting time of a new job arriving to an active cluster. It would also make it possible to compare the predicted waiting time against the built in *Slurm* Start Time estimator directly.
5. **Exploration of Changing Resource Availability:** Extending the applicability of the SimPE model in different settings, such as with dynamically changing resource availability, will help in understanding its adaptability. This exploration is important for ensuring the model's relevance in various real-world computing infrastructures.
6. **Evaluation of Other Slurm Simulators:** A thorough examination of other existing Slurm simulators that are based on the Slurm source code is recommended. This evaluation should focus on identifying simulators that support a wider range of scheduling policies.

Understanding the capabilities and limitations of these simulators can provide valuable insights and potentially enhance the functionality of the SimPE model. Alternatively, it could be an alternative tool for scenario analysis or waiting time estimation.

7. **Development of Multi-Resource Models:** Building more detailed multi-resource models will address the current limitations of the PE-focused approach. This development is important for a more nuanced simulation of complex real-world computing clusters, thereby providing a deeper understanding of cluster dynamics.
8. **Adaptation for Varied Use Cases:** Adapting the SimPE model for varied use cases, such as estimating a spot price for different groups on the cluster similar to a Amazon spot price. Alternatively, the estimated waiting time could be compared against the estimated waiting time on a cloud computing provider. These adaptations could provide valuable insights into the economic and scalability aspects of resource allocation.

7.4 Code Availability

The code developed for this thesis, including the SimPE model implementation, configuration files, and simulation scripts, is freely available under the MIT License at https://github.com/Overlytic/SimPE_validation. This permissive license allows for both academic and commercial use of the code, with the only requirement being the inclusion of the original copyright notice and license terms. The repository contains anonymised configuration files and documentation to help reproduce the results presented in this work.

References

- Abdi, Hervé (2010). “Coefficient of variation”. In: *Encyclopedia of Research Design* 1.5, pp. 169–171.
- Aida, Kento (2000). “Effect of job size characteristics on job scheduling performance”. In: *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer, pp. 1–17.
- Alvarez, Gonzalo Pedro Rodrigo et al. (2016). “Towards Understanding Job Heterogeneity in HPC: A NERSC Case Study”. In: *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*. IEEE, pp. 521–526.
- Araya, Mauricio et al. (2018). “JOVIAL: Notebook-based astronomical data analysis in the cloud”. In: *Astronomy and Computing* 25, pp. 110–117.
- Baker, Mark and Rajkumar Buyya (1999). “Cluster computing at a glance”. In: *High Performance Cluster Computing: Architectures and Systems*. Ed. by Rajkumar Buyya. Vol. 1. Prentice Hall, pp. 3–45.
- Bauer, Amanda E. et al. (2019). “Petabytes to science”. In: *arXiv preprint arXiv:1905.05116*.
- Bogert, F. Alexander et al. (2014). “Computing in astronomy: applications and examples”. In: *Computer* 47.9, pp. 62–69.
- Bose, Sanjay K. (2013). *An introduction to queueing systems*. Springer Science & Business Media. DOI: [10.1007/978-1-4615-0001-8](https://doi.org/10.1007/978-1-4615-0001-8).
- Bottum, Jim et al. (2017). *The Future of Cloud for Academic Research Computing: Results of an NSF-Supported Workshop, Entitled “Cloud Forward”*. Tech. rep. NSF ACI/CSE Award 1632037.
- Bourke, Tyler et al. (2015). “Advancing astrophysics with the Square Kilometre Array”. In: *Proceedings of Science (AASKA14)*. ITA, p. 215.
- Brederode, Leonardus R et al. (2016). “MeerKAT: a project status report”. In: *Ground-based and Airborne Telescopes VI*. Vol. 9906. SPIE, pp. 764–778.
- Brian, Hayes et al. (2008). “Cloud computing”. In: *Communications of the ACM* 51.7, pp. 9–11.
- Broekema, P. Chris, Rob V van Nieuwpoort, and Henri E. Bal (2015). “The Square Kilometre Array science data processor preliminary compute platform design”. In: *Journal of Instrumentation* 10.07, p. C07004.
- Broekema, P. Chris, Rob V Van Nieuwpoort, and Henri E Bal (2012). “Exascale high performance computing in the Square Kilometer Array”. In: *Proceedings of the 2012 workshop on High-Performance Computing for Astronomy Data*, pp. 9–16.
- Brown, Nick et al. (2022). “Predicting batch queue job wait times for informed scheduling of urgent HPC workloads”. In: *arXiv preprint arXiv:2204.13543*.
- Byrne, Ruby and Danny Jacobs (2021). “Development of a high throughput cloud-based data pipeline for 21 cm cosmology”. In: *Astronomy and Computing* 34, p. 100447.

- D’Isanto, Antonio and Kai Lars Polsterer (2018). “Photometric redshift estimation via deep learning-generalized and pre-classification-less, image based, fully probabilistic redshifts”. In: *Astronomy and Astrophysics* 609, A111.
- DeBoer, David R et al. (2017). “Hydrogen epoch of reionization array (HERA)”. In: *Publications of the Astronomical Society of the Pacific* 129.974, p. 045001.
- Downey, A. B. (Apr. 1997). “Predicting queue times on space-sharing parallel computers”. In: *Proceedings 11th International Parallel Processing Symposium*, pp. 209–218. DOI: [10.1109/IPPS.1997.580894](https://doi.org/10.1109/IPPS.1997.580894).
- Erlang, Agner Krarup (1909). “The theory of probabilities and telephone conversations”. In: *Nyt. Tidsskr. Mat. Ser. B* 20, pp. 33–39.
- Fanaroff, BL (2022). “The Square Kilometer Array and a Golden Age for Astronomy”. In: *Resonance* 27.4, pp. 529–542.
- Feitelson, Dror G. and Larry Rudolph (1998). “Metrics and benchmarking for parallel job scheduling”. In: *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer, pp. 1–24.
- Frachtenberg, Eitan and Dror G. Feitelson (2005). “Pitfalls in parallel job scheduling evaluation”. In: *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer, pp. 257–282.
- Glass, I. S. (2020). “Radio astronomy in South Africa”. In: *Monthly Notes of the Astronomical Society of South Africa* 79, pp. 162–172.
- Grainge, Keith et al. (2017). “Square Kilometre Array: The radio telescope of the XXI century”. In: *Astronomy Reports* 61, pp. 288–296.
- Ham, Ruud van der (2018). “Salabim: discrete event simulation and animation in python”. In: *Journal of Open Source Software* 3.27, p. 767.
- Heywood, I et al. (2022). “The 1.28 GHz MeerKAT galactic center mosaic”. In: *The Astrophysical Journal* 925.2, p. 165.
- Huling, Derek and Scott B. Miles (2015). “Simulating disaster recovery as discrete event processes using python”. In: *2015 IEEE Global Humanitarian Technology Conference (GHTC)*. IEEE, pp. 248–253.
- Islam, Rafia et al. (2023). “The future of cloud computing: benefits and challenges”. In: *International Journal of Communications, Network and System Sciences* 16.4, pp. 53–65.
- Jackson, David, Quinn Snell, and Mark Clement (2001). “Core algorithms of the Maui scheduler”. In: *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer, pp. 87–102.
- Javadi, Bahman, Ruppa K. Thulasiram, and Rajkumar Buyya (2013). “Characterizing spot price dynamics in public cloud environments”. In: *Future Generation Computer Systems*. Special Section: Utility and Cloud Computing 29.4, pp. 988–999. DOI: [10.1016/j.future.2012.06.012](https://doi.org/10.1016/j.future.2012.06.012).
- Johnston, Simon et al. (2007). “Science with the Australian Square Kilometre Array Pathfinder”. In: *Publications of the Astronomical Society of Australia* 24.4, pp. 174–188.
- Jokanovic, Ana, Marco D’Amico, and Julita Corbalan (Nov. 2018). “Evaluating slurm simulator with real-machine slurm and vice versa”. In: *2018 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, pp. 72–82. DOI: [10.1109/PMBS.2018.8641556](https://doi.org/10.1109/PMBS.2018.8641556).
- Jonas, Justin and MeerKAT Team (2016). “The MeerKAT radio telescope”. In: *MeerKAT Science: On the Pathway to the SKA*, p. 1.

- Joubert, AF, Bruce Merry, SKA SA, et al. (2017). “Orchestrating MeerKAT’s distributed science data processing pipelines”. In: *Proceedings of the 16th International Conference on Accelerator and Large Experimental Control Systems (ICALEPCS2017)*. JACoW Publishing. Chap. IT Infrastructure for Control Systems, pp. 1152–1157. DOI: [10.18429/JACoW-ICALEPCS2017-THBPA04](https://doi.org/10.18429/JACoW-ICALEPCS2017-THBPA04).
- Jozsa, Gyula IG et al. (2022). “The detection of a massive chain of dark H i clouds in the GAMA G23 field”. In: *The Astrophysical Journal* 926.2, p. 167.
- Kendall, David G. (1953). “Stochastic processes occurring in the theory of queues and their analysis by the method of the imbedded markov chain”. In: *The Annals of Mathematical Statistics*, pp. 338–354.
- Khalil, M et al. (2019). “Big data in astronomy: from evolution to revolution”. In: *International Journal of Advanced Astronomy* 7.1, pp. 11–14.
- Kingman, J. F. C. (1961). “The single server queue in heavy traffic”. In: *Mathematical Proceedings of the Cambridge Philosophical Society*. Vol. 57. 4. Cambridge University Press, pp. 902–904.
- Kingman, John FC (2009). “The first Erlang century—and the next”. In: *Queueing Systems* 63.1-4, p. 3.
- Klusáček, Dalibor and Hana Rudová (2014). “Multi-Resource Aware Fairsharing for Heterogeneous Systems”. In: *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer, pp. 53–69.
- Li, Hui et al. (2004). “Predicting job start times on clusters”. In: *IEEE International Symposium on Cluster Computing and the Grid, 2004. CCGrid 2004*. IEEE, pp. 301–308.
- Li, Zheng et al. (2013). “Early observations on performance of Google compute engine for scientific computing”. In: *2013 IEEE 5th International Conference on Cloud Computing Technology and Science*. Vol. 1. IEEE, pp. 1–8.
- Lim, Seung-Hwan et al. (2012). “D-factor: a quantitative model of application slow-down in multi-resource shared systems”. In: *ACM SIGMETRICS Performance Evaluation Review* 40.1, pp. 271–282.
- Lucero, Alejandro (2011). “Simulation of batch scheduling using real production-ready software tools”. In: *Proceedings of the 5th IBERGRID*, pp. 21–92.
- Mao, Yishu et al. (2016). “DAG constrained scheduling prototype for an astronomy exascale HPC application”. In: *2016 IEEE 18th International Conference on High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems (HPC-C/SmartCity/DSS)*, pp. 631–638. DOI: [10.1109/HPCC-SmartCity-DSS.2016.0094](https://doi.org/10.1109/HPCC-SmartCity-DSS.2016.0094).
- Matloff, Norm (2008). “Introduction to discrete-event simulation and the simpy language”. In: *Davis, CA. Dept of Computer Science* 2.2009, pp. 1–33.
- Merwe, Helga van der et al. (2021). “Developing an environmental research platform in the Karoo at the Square Kilometre Array”. In: *South African Journal of Science* 117.11-12, pp. 1–7.
- Murali, Prakash and Sathish Vadhiyar (2016). “Qespera: an adaptive framework for prediction of queue waiting times in supercomputer systems”. In: *Concurrency and Computation: Practice and Experience* 28.9, pp. 2685–2710.
- Onose, Alexandru et al. (2016). “Scalable splitting algorithms for big-data interferometric imaging in the SKA era”. In: *Monthly Notices of the Royal Astronomical Society* 462.4, pp. 4314–4335.

- Portella, Gustavo et al. (2019). “Statistical analysis of Amazon EC2 cloud pricing models”. In: *Concurrency and Computation: Practice and Experience* 31.18, e4451.
- Prabhu, CSR (2008). *Grid and Cluster Computing*. PHI Learning Pvt. Ltd.
- Qian, Ling and Zhiguo Luo (2009). “Cloud computing: An overview”. In: *IEEE International Conference on Cloud Computing*, pp. 626–631.
- Rajwade, KM et al. (2022). “First discoveries and localizations of Fast Radio Bursts with MeerTRAP: real-time, commensal MeerKAT survey”. In: *Monthly Notices of the Royal Astronomical Society* 514.2, pp. 1961–1974.
- Rodrigo, Gonzalo P et al. (2017). “SCSF: A scheduling simulation framework”. In: *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer, pp. 152–173.
- Roy, Arnab, Joseph Lalnunfela Pachuau, and Anish Kumar Saha (2021). “An overview of queuing delay and various delay based algorithms in networks”. In: *Computing* 103.10, pp. 2361–2399.
- Simakov, Nikolay A et al. (2017). “A slurm simulator: Implementation and parametric analysis”. In: *International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems*. Springer, pp. 197–217. DOI: [10.1007/978-3-319-72971-8_10](https://doi.org/10.1007/978-3-319-72971-8_10).
- Simakov, Nikolay A. et al. (2022). “Developing accurate Slurm Simulator”. In: *Practice and Experience in Advanced Research Computing*, pp. 1–4.
- Smith, Arfon M et al. (2019). “Astronomy should be in the clouds”. In: *Bulletin of the American Astronomical Society* 51.7, p. 55.
- Smith, Warren, Valerie Taylor, and Ian Foster (1999). “Using run-time predictions to estimate queue wait times and improve scheduler performance”. In: *Workshop on Job scheduling strategies for Parallel Processing*. Springer, pp. 202–219.
- Soysal, Mehmet, Marco Berghoff, and Achim Streit (2018). “Analysis of job metadata for enhanced wall time prediction”. In: *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer, pp. 1–14. DOI: [10.1007/978-3-030-10632-4_1](https://doi.org/10.1007/978-3-030-10632-4_1).
- Sterling, Thomas, Maciej Brodowicz, and Matthew Anderson (2017). *High Performance Computing: Modern Systems and Practices*. Morgan Kaufmann.
- Stewart, William J. (2009). *Probability, Markov Chains, Queues, and Simulation: the Mathematical Basis of Performance Modeling*. Princeton University Press. DOI: [10.2307/j.ctvcn4gtc](https://doi.org/10.2307/j.ctvcn4gtc).
- Stewart Robinson, Simulation (2004). *The practice of model development and use*.
- Sundarapandian, Vaidyanathan (2009). *Probability, Statistics and Queuing Theory*. PHI Learning Pvt. Ltd.
- Sunyaev, Ali (2020). “Cloud computing”. In: *Internet Computing*. Springer, pp. 195–236.
- Swart, Gerhard P, Peter E Dewdney, and Andrea Cremonini (2022). “Highlights of the SKA1-Mid telescope architecture”. In: *Journal of Astronomical Telescopes, Instruments, and Systems* 8.1, pp. 011021–011021.
- Tang, Wei et al. (2009). “Fault-aware, utility-based job scheduling on blue, gene/p systems”. In: *2009 IEEE International Conference on Cluster Computing and Workshops*, pp. 1–10. DOI: [10.1109/CLUSTER.2009.5289206](https://doi.org/10.1109/CLUSTER.2009.5289206).
- Taylor, AR (2012). “The Square Kilometre Array”. In: *Proceedings of the International Astronomical Union* 8.S291, pp. 337–341.
- Taylor, Greg B, Chris Luke Carilli, and Richard A Perley (1999). “Synthesis imaging in radio astronomy II”. In: *Synthesis Imaging in Radio Astronomy II* 180.
- Tingay, SJ et al. (2012). “Realisation of a low frequency SKA precursor: the Murchison Widefield Array”. In: *arXiv preprint arXiv:1212.1327*.

Trofinoff, Stephen and Massimo Benini (2015). “Using and modifying the BSC slurm workload simulator”. In: *Slurm User Group Meeting*.

Appendix A

Appendix: SimPE Modelling Detail

This appendix continues the discussion of the detail of the SimPE modelling code that was started in Section 4.4. In particular, showing a shortened version of the *Slurm_scheduler* process in Listing A.1 and the code for setting up the *Salabim* environment and running the SimPE model in Listing A.2.

```
1
2 class Slurm_scheduler(sim.Component):
3     def process(self):
4         while True:
5             run_again = False
6
7             # 1. Check available PE
8             avail_pe = res.available_quantity()
9
10            # 2. Read Queues into dataframes
11            a_df = activeQ_to_df(activeQ)
12            q_df = pendingQ_to_df(pendingQ)
13
14            # 3. If QUEUE_TYPE == "Priority2": Recalculate Job priorities
15            # 4. Find the highest priority job
16            q_df = q_df.sort_values(SORT_COLUMNS, ascending=SORT_ASC)
17            pop_index = q_df.index[0].tolist()
18            pe_req = pendingQ[pop_index].pe
19
20            # 5. If QUEUE_TYPE == "Backfill_Prio": Perform Job Backfill
21            # 6. Run Job
22            if pe_req <= avail_pe:
23                self.job = pendingQ.pop(pop_index)
24                self.job.slurm_comment = slurm_comment
25                self.job.activate()
26
27            # 7. Repeat process if enough pe left to run another job
28            q_df = q_df.drop(index=pop_index)
29
30            if count_eligible_jobs(q_df, pe_req) > 0:
31                run_again = True
32
```

```

33         if not run_again:
34             yield self.passivate()

```

Listing A.1: Python code for the Slurm_scheduler process of the SimPE model

```

1
2  # Salabim Environment
3  env = sim.Environment()
4
5  # Processes
6  slurm = Slurm_scheduler()
7  JobReader(job_batch_list=job_batch_list)
8
9  # Scheduling Parameter
10 QUEUE_TYPE = "Priority"
11
12 # Queue Sorting
13 if QUEUE_TYPE == "FIFO":
14     SORT_COLUMNS = ["job_entertime"]
15     SORT_ASC = [True] # Lowest job enter time
16
17 elif QUEUE_TYPE in ["Priority", "Priority2",
18                     "Backfill", "Backfill_Prio"]:
19
20     SORT_COLUMNS = ["job_priority", "job_entertime"]
21     SORT_ASC = [False, True] # Highest priority, Lowest Job enter time
22
23 else:
24     print("Please set the QUEUE_TYPE variable")
25     SORT_COLUMNS = []
26     SORT_ASC = []
27
28 # Run SimPE model
29 env.run()

```

Listing A.2: Python code for running the SimPE model

Appendix B

Appendix: Slurm Training Cluster Configuration

This appendix shows the main configuration file (*slurm.conf*) for the *Slurm Training cluster*. The file below shows the configuration for the base run with FIFO scheduling. However, under the 'Scheduling' section in the below listing, we also show the configurations for the other scheduling type, namely: Static Priority, Dynamic Priority and Backfill.

```
1 ClusterName=tinus_testing
2
3 SlurmctldHost=tinus-controller
4 MpiDefault=none
5 ProctrackType=proctrack/cgroup
6 RebootProgram="/sbin/reboot"
7 ReturnToService=1
8 SlurmctldPidFile=/var/run/slurmctld.pid
9 SlurmctldPort=6817
10 SlurmdPidFile=/var/run/slurmd.pid
11 SlurmdPort=6818
12 SlurmdSpoolDir=/var/spool/slurmd
13 SlurmUser=slurm
14 StateSaveLocation=/var/spool/slurm_state
15 SwitchType=switch/none
16 TaskPlugin=task/affinity,task/cgroup
17
18 # TIMERS
19 InactiveLimit=0
20 KillWait=30
21 MinJobAge=3600
22 SlurmctldTimeout=120
23 SlurmdTimeout=60
24 Waittime=0
25
26 # RESOURCES
27 SelectType=select/cons_res
28 SelectTypeParameters=CR_CPU_Memory
29
```

```

30 ##### SCHEDULING #####
31
32 ### FIFO:
33
34 SchedulerType=sched/builtin
35 PriorityType=priority/multifactor
36
37 ### Static Priority:
38
39 # SchedulerType=sched/builtin
40 # PriorityType=priority/multifactor
41 # PriorityWeightAssoc=10000
42
43 ### Dynamic Priority
44
45 # SchedulerType=sched/builtin
46 # PriorityType=priority/multifactor
47 # PriorityWeightAssoc=10000
48 # PriorityWeightAge=500
49 # PriorityMaxAge=3 #10 minutes.
50
51 ### Backfill
52
53 # SchedulerType=sched/backfill
54 # PriorityType=priority/basic
55
56 # OTHER PRIORITY SETTINGS
57 PriorityDecayHalfLife=0
58 PriorityUsageResetPeriod=YEARLY
59 PriorityCalcPeriod=1
60
61 # LOGGING AND ACCOUNTING
62 AccountingStorageEnforce=limits,qos
63 AccountingStorageHost=tinus-database
64 AccountingStoragePort=7031
65 AccountingStorageType=accounting_storage/slurmdbd
66 AccountingStorageUser=slurm
67
68 JobCompHost=training_controller
69
70 JobCompType=jobcomp/none
71 JobAcctGatherFrequency=30
72 JobAcctGatherType=jobacct_gather/cgroup
73
74 DebugFlags=Backfill,BackfillMap,SelectType
75 SlurmctldDebug=debug
76 SlurmctldLogFile=/var/log/slurm/slurmctld.log
77 SlurmdDebug=debug
78 SlurmdLogFile=/var/log/slurm/slurmd.log
79 SlurmSchedLogFile=/var/log/slurm/slurmsched.log
80
81 # COMPUTE NODES
82 NodeName=tinus-compute NodeAddr=192.168.0.14 CPUs=8 RealMemory=59392
   Sockets=8 CoresPerSocket=1 ThreadsPerCore=1 State=UNKNOWN
83
84 PartitionName=training Nodes=tinus-compute Default=YES MaxTime
   =14-00:00:00 DefMemPerCPU=6144 State=UP

```

Listing B.1: slurm.conf for the Slurm Training Cluster