UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

FIB

EVALUATING THE IMPACT OF TASK AGGREGATION IN WORKFLOWS WITH SHARED RESOURCES ENVIRONMENTS

MANUEL GIMÉNEZ DE CASTRO MARCIANI

Thesis supervisor: MIGUEL CASTRILLO (Barcelona Supercomputing Center)

Thesis co-supervisor: MARIO CESAR ACOSTA COBOS (Department of Computer Architecture)

Tutor: GLADYS MIRIAM UTRERA IGLESIAS (Department of Computer Architecture)

Degree: Master Degree in Innovation and Research in Informatics (Advanced Computing)

Master's thesis

Facultat d'Informàtica de Barcelona (FIB)

Universitat Politècnica de Catalunya (UPC) - BarcelonaTech

A Papá,

que me inspiró a cruzar el charco.

Abstract

We study the relative impact of task aggregation, or wrapping, which is a technique meant for computational workflows that bundles jobs into a single submission to be sent to remote schedulers. Experiments inside the Earth Science community can be lengthy and compriseseveral steps with many dependencies. The community has traditionally focused in increasing the performance of the models, but the overall execution of the workflow, including the queue time, has received little interest. Aiming to reduce the time spent in queue, the developers of Autosubmit, a workflow manager developed for climate simulations, weather forecast simulations, and air quality simulations, came up with task aggregating, or wrapping. Our objective is to assess if this technique does indeed reduce the total queue time of the workflow. The complex interplay between the dynamic nature of the usage of the machine and the scheduler policy plays a central role in our analysis, which poses the main challenge of this work. Hence, we do an intricate study of the scheduling policy of the popular Slurm Workload Manager and a statistical characterization of the usage of both simulated machines: LUMI and CEA-Curie. With that, we perform a twofold experimentation: a simulation using dynamic workloads – where job arrival time plays a role – with a workflow composed of multiple jobs and a static workload – where all jobs in the workload are submitted at the same time – varying job and user factors that play a role into the scheduling. Results show that aggregation is beneficial in the majority of cases for the workflows that are vertically organized – that is, a chain of submissions where each job is dependent on the previous –, whilst for the horizontal arranged workflows – where jobs do not have dependencies – it might undermine the queue time depending on the user's past usage and the machine's current state.

Contents

List of Figures

List of Tables

Glossary

climate multidecadal simulations are climate simulations with a time horizon of a few decades

climate simulations are NWP simulations to predict the future state of the climate system for time horizons ranging several weeks, months, or years using information from the past and current state of the system. They bridge the gap between weather forecast simulations for the coming days and climate projections up to the end of the 21st century

Curie was a cluster operated by TGCC by CEA from 2011 to 2017

data assimilation is the science of combining different sources of information to estimate the state of a system

operational toolchains are predictions made on using the output from climate simulations to make application specific predictions

reanalysis is the combination past short-range weather forecast simulations with observations through data assimilation

weather forecast simulations are NWP simulations to predict the future state of the system up to a few days based on past and present data

Acronyms

AEMET *Agencia Estatal de Meteorología*

AIC Akaike Information Criterion

ALCF Argonne Leadership Computing Facility

BSC Barcelona Supercomputing Center

CAMS Copernicus Atmosphere Monitoring Service

CEA *Commissariat à l'énergie atomique et aux énergies alternatives*

CMIP Coupled Model Intercomparison Project

CPMIP Computational Performance Model Intercomparison Project

EC-Earth European Consortium for Earth System Model

ECMWF European Centre for Medium-Range Weather Forecasts

ICON Icosahedral Nonhydrostatic Weather and Climate Model

IFS Integrated Forecasting System

LLNL Lawrence Livermore National Laboratory

MONARCH Multiscale Online Nonhydrostatic Atmosphere Chemistry Model

NEMO Nucleus for European Modelling of the Ocean

NWP Numerical Weather Prediction

SSE Sum of Squared Errors

SWF Standard Workflow Format

TGCC *Très Grand Centre de Calcul du CEA*

UBCCR University of Buffalo's Center for Computational Research

Chapter 1

Introduction

1.1 Motivation

Since the dawn of the massively parallel supercomputers, by the end of the 20th century, we witnessed a considerable increase of the demand for computational resources in all sorts of areas. To name a few: climate sciences [1], Numerical Weather Prediction, engineering [2], and bioinformatics [3]. These large cluster systems composed of – ever so increasing number of – independent machines working in parallel, sometimes made of consumer grade CPUs, got popular starting with Intel's iPSC in 1985. Some notable examples are the MareNostrum family at the Barcelona Supercomputing Center which its first version was set up in March of 2004 [4]; Blue Gene [5], introduced in 2005; or Riken's K supercomputer [6], introduced in 2012.

To manage the usage of these parallel resources by multiple users, a special software was created. This software, called batch scheduler, is responsible for queuing, allocating, and dealing with fault tolerance. Users need only to define their task, or job, which is, the executable, the input data, and the resources to run it. The batch scheduler handles the allocation and, afterwards, the execution. With enough users aspiring for the resources of the system, we have a competitive resource environment.

At the beginning, when the requested resources exceeds the available, schedulers applied a simple first-in-first-out policy. But in the last few years, in order to address the increase of the demand for parallel resources and looking to increase the efficiency – in various different ways, like throughput [7], energetic efficiency [8], or IO [9] – of the machine, scheduler developers came up with more intricate queuing algorithms. For example, Slurm Workload Manager [10] – a popular batch scheduler used in MareNostrum 4 [11], LUMI [12], Levante [13], Meluxina [14], and many others HPC centers – allows system administrators to manage the queuing of jobs by taking into account multiple factors, where the most important are: allocated CPUs, age, partition, QoS, and fair share.

The fair share factor, in fact, is one of the solutions that have been explored since the 80s from the landmark paper by Kay and Lauder [15], which aimed to balance the usage of the mainframe computer shared amongst students of the university of Sydney for learning purposes. That paper implemented fairness in the context of the scheduling of processes within the operating system. Once students did not need to share resources anymore because each of them got their own computer, this technique was deemed unnecessary. However, with the new supercomputer access given to more people, Kay and Lauder's solution reemerged to balance the system resources among all the users. Slurm implements it in the same way as proposed by them, called *Classic*, or another more sophisticated way, called *Fair Tree*.

Moreover, the necessity of more accurate queue time predictions increased with the number of users in such platforms [16]. From the user's point of view this necessity is obvious: a job could be appropriately tailored to execute and meet with deadlines. From the scheduler's point of view, queue time prediction is useful, for example, for backfill algorithms [7] in order to increase the throughput of the machine. These algorithms execute queuing jobs with the premise of not interfering with the execution of the jobs in ahead in the queue. Usually, such scenarios occur when a machine is emptying itself to allow a large job to be run, which creates spaces that smaller jobs can utilize.

But queue prediction, even for the simplest queuing algorithm, is by no means trivial since the *time spent in queue is dependent on which jobs are in that queue*, and that is governed by the HPC's users behavior. For example, in figure 1.1 we see the evolution of the available and used CPUs on MareNostrum 4 and jobs in queue and running. The plot with the running jobs – which is the one to the right – is highly variable, with peaks of jobs in the queue reaching from a bit less than 500 to up to 2000 jobs. As for the machine utilization – which is the left plot – we see how the total number of available and used resources also vary quite a lot but the machine is always used in almost its entirety. Additionally, administrative decisions profoundly impact the scheduling, because the scheduler can invert its behavior by changing a single flag in the scheduler configuration.

Since the policies impact so much the scheduling, it became a necessity to create faithful representations of the workload, which is the log of execution of the machine. Such logs could be used on simulators by system administrators and researchers to test configurations before going to the operational environment. Many authors [17] [18] [19] have proposed probability distributions to model certain aspects of the workload, like allocated CPUs, runtime, and inter-arrival time. With appropriate parameters for these distributions, they could generate workloads, which are called *synthetic workloads*, that behave similarly to the observed data.

With modelling in mind, Chapin, Cirne, and Feitelson [20] have proposed a format to describe workloads in an standard anonymized way. These workloads, or traces, are available at Feitelson's repository [21] so that researchers and system administrators can use them as testing bench. But unfortunately, managing institutions are in general reluctant to give data even in this anonymized way which *makes modelling workloads de facto impossible*. It would be beneficial, for the both the scheduling community as to anyone interested in modelling HPC environments, to have more raw data published or – at the very least – more papers on the characterization of the workloads of newer large HPC systems.

As for the user's side, in the Earth Science community there is a growing interest in increasing the computational performance of *the whole simulation*, accounting not only for the runtime of each of the tasks, but also their queue time. An example of such works is Acosta et al. [22] where the authors investigated the performance of CMIP6 experiments under the CPMIP project, which aims for an unified set of measurements for Earth System models accounting for more factors than traditionally it has been done: such as energy, queue time, interruptions, etc. Authors found performance degradation of up to 50% on these simulations due to time spent on queue and/or interruptions.

Operationally, optimizing the queuing times makes a lot of sense, since scientists and technicians have to deliver their simulation results with strict deadlines. AEMET, the Spanish meteorological agency, has its own resources to run their forecasts, but research centers like the Barcelona Supercomputing Center share their computing resources – even though in the Earth Science department there are also operational simulations running –. And, even for climate multidecadal simulations runs, scientists need to meet

project deadlines and a certain amount of simulated years. Hence, the community is exploring ways of optimizing *not only the execution* per se of the model, by choosing the optimal number of CPUs, but also the *interaction that those choices have with the scheduler*. And, in this work, *we aim for the latter.*

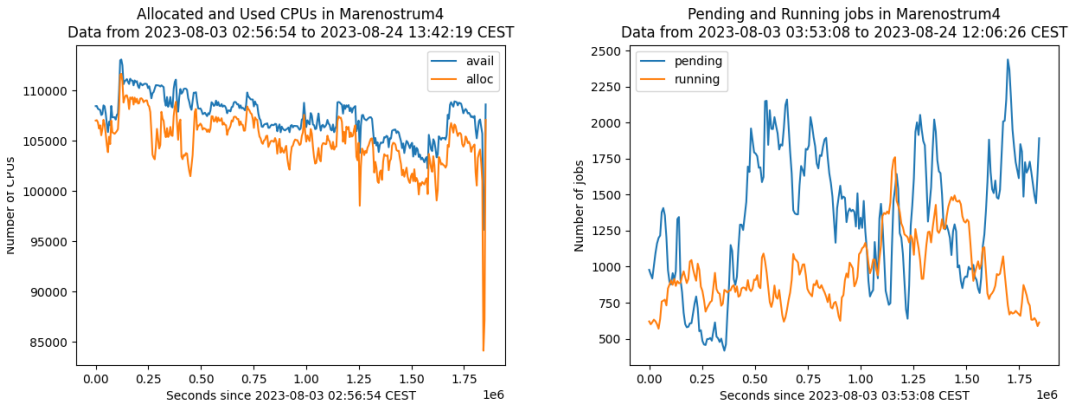

Figure 1.1: Evolution of the available and used CPUs and running and pending jobs at MareNostrum 4. The data was taken on the 24th of August 2023 from the BSC operation's HPC portal.

1.2 Objective

We aim to reduce the *total time of the execution of the Earth modelling workflows*, that is the runtime *included* the time in queue. For that, we are going to *assess the relative impact of the task aggregation*, or *wrapping*, under different scenarios. Our thesis is that *aggregating tasks when the user has a low share factor is beneficial*, since it would save submitting multiple individual jobs onto a bad scenario. We just take the burden of a "bad" fair share a single time. All of these will be considered under the scheduling policies of the platform we most use and have access to its scheduling policies, MareNostrum 4.

Aggregation – in its elemental types – is depicted in images 1.2 and 1.3, where we have a black box indicating the jobs that are submitted in tandem. Jobs from inside the aggregated package have their dependencies respected. We also have the mix-and-match, in image 1.4, where jobs within the wrapper can run one row per time or two columns concurrently.

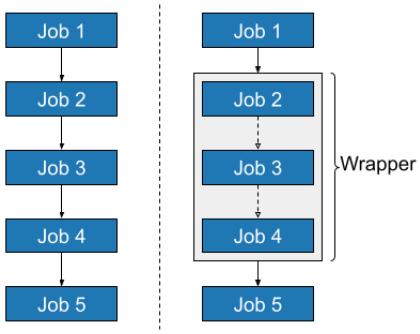

Figure 1.2: Description of the basic mechanism of the task aggregation, or wrapping, for the vertical case. Jobs inside the black box are those that are submitted at the same time to the the HPC platform.

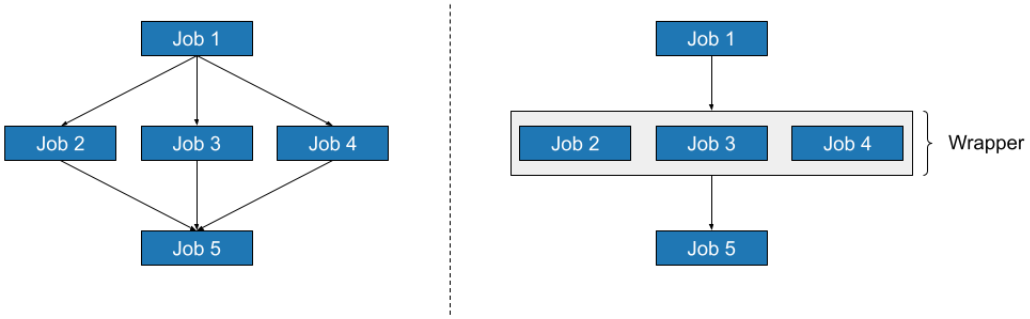

Figure 1.3: Description of the basic mechanism of the task aggregation, or wrapping, for the horizontal case. Jobs inside the black box are those that are submitted at the same time to the the HPC platform.

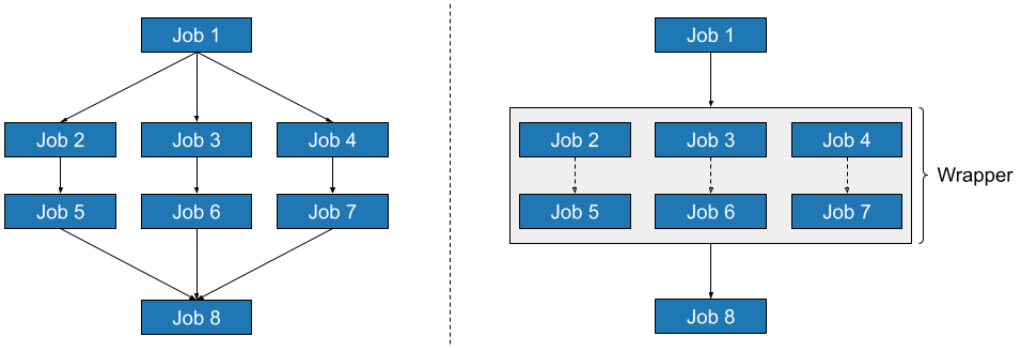

Figure 1.4: Description of the basic mechanism of the task aggregation, or wrapping, for the horizontal-vertical case. Jobs inside the black box are those that are submitted at the same time to the the HPC platform.

1.3 Contribution

In this work, we contribute in the simulator with

- *merging* of both BSC's [23] and UBCCR's [24] Slurm simulator to provide support for user accounting, and, consequently, priority computation via *multifactor*,
- *adapting the source code* of the simulator to expose the priority, so we can keep track of it,
- *implementing* the simulator's *builtin* scheduler[1], which is the name of the scheduler option *without backfill*. Although it still lacks validation, hence its results were ruled out.

As for the workload file, which is fundamental for simulating an HPC machine, we cover all steps for characterizing it for the LUMI supercomputer:

- we *gather* and make a *statistical analysis*,
- we *generate* synthetic workloads following distributions that were fitted to the data.

[1] Available at https://earth.bsc.es/gitlab/mgimenez/ces_slurm_simulator.

The analysis is composed of a first phase of a descriptive analysis and then we followed Feitelson's workload modelling book [18]. Using this static synthetic workloads, we

- *create the software stack* to allow us to reproduce and automatize the simulation process. This meant a novel Docker[2] [25] image for the Slurm simulator,
- *conduct 2000 experiments* varying parameters to quantify their relative impact in waiting time and priority upon start of the execution,
- *propose a model to describe* the relation of fair share and wait time.

We also conduct longer simulations. For that, we analyze the workload from CEA-Curie machine to check its adequacy as a general purpose, resource sharing, and scientific platform like MareNostrum 4. We include two typical workflows on it: one vertical and another horizontal. Hence, we *develop a methodology to control the fair share* upon submission of the tracked workflow.

Finally, we *theorize the relation* between the fair share and the average waiting time. We *confirm the importance* of the fair share in the priority as per using the weights in MareNostrum 4. We *propose strategies* for task aggregation, or wrapping, to gain performance in the overall execution of the workflow. We find that we *need to delve deeper* in the interplay between backfill and low fair share factor value.

[2] Available at `https://earth.bsc.es/gitlab/mgimenez/docker-ubuntu-ces-slurm-sim`.

Chapter 2

Background

In this chapter, we will lay down the main concepts needed for this work. We will start by an overview on the very last that was published, either in peer-reviewed journals or pre-print sharing platforms. After, we will cover the scheduler that we have our simulator built on top of, what is a workload and its types, what is a workflow and how we execute them on HPC platforms, and – finally – the technique we apply to reduce the queue time.

2.1 Related Work

In the literature, most of the papers related with this work are focused in waiting time prediction, mainly for scheduler decisions. These usually devote a small part of their work to the workload modelling that is necessary to test their hypothesis. As for optimizing the workflow execution, most of the work has been done when running it on clouds providers.

Workload modelling is its own discipline in terms of challenges. Many authors [17] [18] [19] have proposed the probability distribution in which a particular job attribute might follow based from empirical data. But for the very last in this area, we have the characterization of jobs done by Patel et al. [26] for the Intrepid and Mira supercomputers from Argonne Leadership Computing Facility. They have showed that in the "past decade resource utilization of leadership class systems at ALCF has increased significantly – almost nearing full utilization most of the time." Moreover, they have shown that the job sizes has increased both in runtime and CPUs allocated.

There were also efforts to build synthetic workloads – that is, generated –. Out the latest, Walfredo Cirne proposed one in 2001 [19]. His work comprises a methodology to generate jobs that follow the distribution of arrival time seen in three different supercomputers. Moreover, he also included the empirically observed distribution of cancelled jobs. For the job size, he used Downey's [17] result and made jobs sizes according to the log uniform distribution.

Out of the newest in queue prediction, Park et al. [16] propose a model to predict congestion of the queue using "features" of the jobs, that is, number of processes, allocated CPUs, etc. They used a hidden Markov to model the queue state. They found that it was very hard to predict queuing time because there are many unknown factors such as the scheduling algorithm and "characteristics of the executed job". For the workload modelling, they have used their institution's managed supercomputer, Nurion, logs.

Machine learning for prediction is extremely popular nowadays, hence there is also a paper by Brown et al. [27] which aim to have accurate starting time using ML for

urgent workflows. They apply K nearest neighbors and boosted trees to predict queue time for 3 HPC machines in the UK under different configurations. Their methodology was able to predict within 1 minute the actual start time about 65% to 76% of the job start times across all the three machines.

As for optimization in queue time, normally the works are carried out to optimize resources in the popular cloud providers. In the work by Kok Konjaang and Xu [28] provide an algorithm to split tasks to reduce their runtime. This is done to optimize both cost and the execution of the whole workflow. Their work reduced the cost by 8%, the total execution by 10% , and improved the resource utilization by 53% of the workflow.

2.2 Slurm Scheduler

Slurm is a free, open-source, fault-tolerant, and highly scalable cluster management and job scheduling system that was developed in a collaborative way between, primarily, Lawrence Livermore National Laboratory, Linux NetworX, Hewlett-Packard and *Groupe* Bull in 2001. In 2010, two members of LLNL created a company to manage the development and marketing of Slurm. Since then, it has been used in many flagship machines.

Slurm is written in C and it was designed with modularity in mind. One example of its modular design is its support for plugins, which are added when the tool is compiled. Over 100 plugins are available, besides ample support for customization. As seen in figure 2.1, Slurm is composed of multiple daemons: *slurmctld*, *slurmd*, *slurmdbd*. The first one is the centralized manager and is responsible for keeping track and orchestrating each node's *slurmd* daemon. In order to configure users, it is necessary to have configured the *slurmdbd* daemon, which is the interface between the manager and the SQL database. Users are configured in tuples of *(user, account, partition, cluster)* which are stored in a SQL database. *slurmd* is the deamon that runs on each node of the cluster and communicates with the main controller the state of the node.

Normally, users are allowed to login only to a few nodes, where they set up their work. Once the data and executables are configured, users prepare their scripts to be run in the machine using special directives in the header: either *sbatch* or *salloc* or launch them with *srun* via command line. In the bare minimum, users are required to specify the number of CPUs, wallclock, and which QoS they choose. Additionally, usually needed for debug, there are some nodes which can be used interactively, which will grant the requested resources as if they were login nodes.

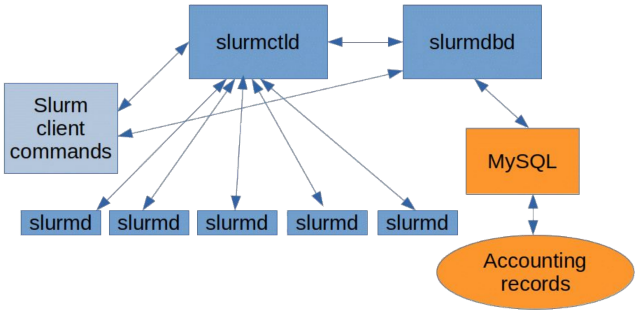

Figure 2.1: Diagram depicting the organization of Slurm and its daemons. Taken from [29].

2.2.1 Scheduling Loop of Slurm

Under default configuration, once a task is submitted, Slurm will try to execute it immediately. If the resources are not available, it is queued. The order in this queue is governed by the *priority* integer. The queue is *decreasingly* ordered according to it and updated periodically. Hence, normally, *larger priority means earlier scheduling*. In section 2.3 we cover how Slurm computes this value.

Slurm's scheduling design is made of two independent algorithms that periodically traverse the queued jobs and try to execute them as seen in figure 2.2. A lock system is implemented to avoid race conditions among these two, which could run concurrently. The main algorithm works by iterating over the ordered queued jobs and tries to schedule them by looking for available nodes to meet the request. Upon the first failure to schedule, it breaks and sleeps. As for the second algorithm, there are two options by default: *builtin* or *backfill*. The first works identically to the main algorithm. The backfill algorithm is a more comprehensive [7], and consequently expensive, technique that allows jobs with less priority, i.e. lower in the queue, to be scheduled before as long as it does not interfere with the start time of the higher positioned job. In figure 2.2 we can see how the execution of the backfill algorithm takes longer. The main goal of this algorithm is to increase the throughput of the system, by exploiting the fragmentation that might occur as Slurm is preparing the machine to schedule large jobs. *backfill* in Slurm *always reserve full nodes*, regardless of the size of the job.

System administrators can tailor the timings which the main scheduler and the secondary run. Most notably for us [30]:

- *default_queue_depth*: configures how many jobs are going to be tested before the either scheduler forfeits
- *defer* tells Slurm to not try to schedule jobs upon submission. As we can see in tables 2.1 and 2.2, respectively in both CEA-Curie and LUMI we see moments of bursts of submissions that would yield too much time the lock of scheduling to the initial try. That would prevent the other algorithms of executing
- *sched_interval* which specifies how frequently the main algorithm will execute

As for the secondary scheduler, we have the following options:

- *bf_resolution* is the time unit that the backfill algorithm will take into account (i.e. if set to 30 seconds, every runtime of the jobs will be rounded to the closest largest multiple of 30 seconds)
- *bf_window* is how far ahead in minutes the algorithm will see when trying to find a spot to backfill. Clearly, this only applies to the backfill scheduler
- *bf_continue* tells the secondary scheduler to continue execution after the lock release, that is the lock to race conditions, and ignore newly submitted jobs

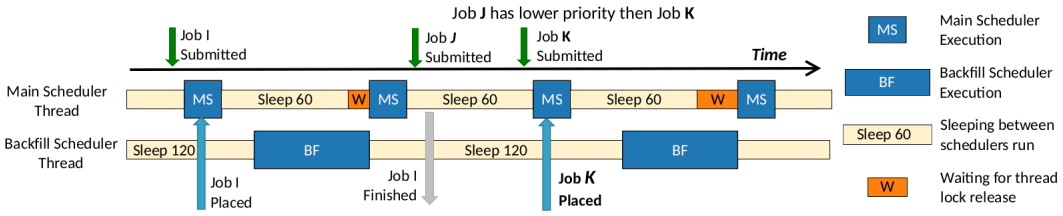

Figure 2.2: Graphical description of the loop logic behind Slurm scheduling. Taken from [24].

Mean	4.50
Std	15.94
Min	0
25 %	0
50 %	1
75 %	4
Max	1414

Table 2.1: Average number of submissions per minute in LUMI during the period February 2023 and June 2023.

Mean	0.86
Std	3.13
Min	0
25 %	0
50 %	0
75 %	1
Max	275

Table 2.2: Average number of submissions per minute in CEA-Curie during the period 1/2/2011 - 17/10/2012.

2.3 Job Priority

Jobs are grouped in decreasing order according to the *priority* integer. On Slurm it can be computed in two ways: *first-in-first-out* and *multifactor*. In FIFO jobs will be given a larger priority the longer they are in queue, while multifactor allows for considering more attributes when deciding which jobs to run. Some of them are: *age*, *size*, *association*, *fair share*, and *quality of service* factors. Factors in Slurm terminology *are always* floats, stored in double precision, between 0 and 1. For each of these values, a 32-bit encoded integer *weight* is associated. The priority is then computed by taking the weighted sum of all factors with their respective weights.

The age factor, which is the one related to the time in queue, starts at 0 and linearly grows until reaching the maximum, which is configured setting the flag *PriorityMaxAge* in the Slurm configuration file. The default is 7 days. Then this factor is computed as

$$a_i = \frac{t}{T}, \tag{2.1}$$

where t is the time elapsed ever since the job submission, discounted the time voluntarily held by user, and T is the total amount of seconds set by administrator with flag

PriorityMaxAge.

By default, the *size* factor is the proportion of CPUs used by the job with respect to the total available. Therefore, when requesting the whole machine, it reaches the maximum. System administrators can invert this by setting the flag *PriorityFavorSmall* to *True*, which would make a single CPU job have the maximum values of 1.0. Hence, the size factor is computed as:

$$s_i = \frac{r_i}{S} \quad \text{or} \quad s_i = \frac{S - r_i + 1}{S}, \tag{2.2}$$

where r_i is the total of CPUs requested by job i and S is the total number of CPUs of the machine.

QoS stands for *Quality of Service*, normally called queues although *they are not separate queues*. It allows users to choose from different options – e.g. standard, debug, xlarge, xlong – depending on their computing needs. Normally they are set up in such a way of awarding the smaller jobs, since by default Slurm gives more priority to larger jobs. From the user side, this means larger priority upon submission, hence less waiting time, at the cost of a more constrained job submission.

Each *QoS* is configured with an associated priority, which is another integer value. Then the factor is computed as the proportion with respect to the largest priority of all QoSs. Then, if the job i was submitted indicating *QoS* q_u, its *QoS* factor is computed as

$$q_i = \frac{q_u}{max_{q \in Q} q}, \tag{2.3}$$

where Q is the set of all *QoSs* set by the system administrator.

The fair share factor is the one that aims to balance the resources among users. It is computed in two different ways, which will be addressed in its separate section 2.4.

Finally, we have the following equation for the priority of a job with respect to the time:

$$P_i(t) = a_i \cdot w_a + s_i \cdot w_s + f_i(t, \Omega(t)) \cdot w_f + q_i w_q, \tag{2.4}$$

where this is the priority at time t of job i. a_i is the age factor, w_a the age factor weight, s_i is the size factor, w_s the weight factor associated with size, f_i is the fair share of the user that launched job i at instant t with machine utilization $\Omega(t)$, w_f is the weight, and finally q_i and w_q are the factor and weight associated to QoS.

2.4 Fair share factor

Fair share factor is a *quantification of a user's right to the machine*. Its objective is twofold: to *seem fair* and to ensure utilization of the machine. The classic implementation, and the original motivation for it, is in paper by Kay and Lauder [15]. But in the following years, there were several proposals with different notions of *what seems fair* and, subsequently, how to implement [31] [8] [32].

Slurm offers two ways of computing the fair share factor: *Classic* and *Fair Tree*. By default, *Fair Tree* is the algorithm used. To prioritize the newer usage of users, Slurm also allows to update the historical usage with a decay factor which is defined by the flag *PriorityDecayHalfLife*, which means the usage will halve every that set amount of time.

In both algorithms, users are assigned an integer value called *RawShares* – their entitlement for the machine – and their usage is tracked, which is called *RawUsage*. This is depicted n figure 2.3, taken from [15].

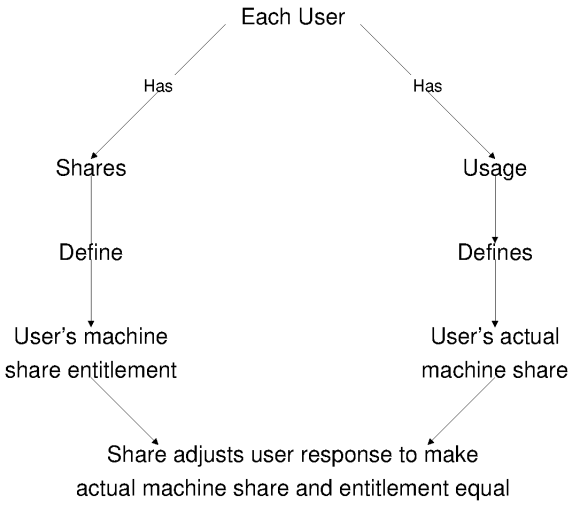

Figure 2.3: Image taken from [15] depicting the basic mechanism behind fair share.

Fair share: Classic Algorithm

Under the Classic algorithm, in order to compute the factor, Slurm normalizes both *RawUsage* and *RawShare* with respect to the total system usage and shares. Then the fair share factor for the user is computed as

$$f_i = 2^{-\frac{ru_u/t_u}{rs_u/t_s}}, \tag{2.5}$$

where ru_u and $t_u = \sum_{j \in U} ru_j$ are the user's *RawUsage* and the system's sum of *RawUsage* and rs_u and $t_s = \sum_{j \in U} rs_j$ are the user's *RawShare* and system's sum of of *RawShares*.

When *RawUsage* is 0, this factor is 1. If the user has exactly the same amount of usage ratio with respect to the share assigned to him, it will be 0.5. Afterwards, it will exponentially decay to 0. This is a major drawback which could cause underflow if a large enough total usage is reached and the user has no usage.

Fair share: Fair Tree Algorithm

The main concept in which *Fair Tree* is built around is to allow the *sharing of the responsiveness* among users in the same account. As the developers put it [33], "users such that if accounts A and B are siblings and A has a higher fair share factor than B, all children of A will have higher fair share factors than all children of B".

When users are added to the Slurm account database, they are included as a tuple which contains: *user*, *account*, *partition*, and *cluster*, where the account is the group that the user is associated with – normally projects that the user is part of –. Users and accounts can be associated to accounts, but accounts can never be associated to users. Plus, slurm does not allow for cycles in the association – e.g. accounts associated with themselves –. Consequently, this structure defines a tree where its root is the special *root* account, which is the only one not child of any account, and every user is a leaf.

The algorithm starts traversing the tree via the *root* account. Every time it stumbles upon an account, it orders decreasingly its children – can be both users and accounts – by their *Level Fair Share*, which is computed by

$$L = \frac{rs_u}{ru_u} \quad \text{or} \quad L = \frac{rs_a}{ru_a}, \tag{2.6}$$

where rs_u and ru_u are the user's *RawShare* and *RawUsage*. It is done likewise if it is an account, where rs_a and ru_a are the account's *RawShare* and *RawUsage*. This value may be infinite if the user or account has no usage. The algorithm traverses this children ordered array. If it finds a user, it assigns a fair factor to it, which is the order in which the algorithm visited the user; if it finds an account, the algorithm recursively continues on it.

We could see this algorithm as assigning fair share factor to users according to an ordering that respects a single rule recursively: no user from an underserved account is higher positioned of another account. Once we have the users ordered, we will assign them the fair share factor according to their position in this array: the first receives a fair share factor of 1.0, the next $1.0 - \frac{1}{n}$ and so on and so forth, where n is the number of *users with different Level Fair Share*. This is because, in case of draw in the Level Fair Share, the algorithm *merges accounts or it assigns the same fair share value to the users*.

Besides allowing for sharing of the fair share among users of the same account, it also deals with the underflow issue that the Classic approach has if the user has a very tiny usage compared with the total system usage.

2.5 Workloads

In the literature, there is an ambiguity with the concept of workload. Sometimes, authors refer to *an application*'s workload. In this work, we are interested in the whole machine use. Hence, will define *workload* as the *full description of jobs* that are run on a machine. In its bare minimum, they include how many CPUs were requested, when they were submitted, started running, and ended. They can be *real*, those that were recorded on machines, or *synthetic*, which were generated. When testing the scheduler, a good representation of the usage of the HPC platforms – the *workload* – is essential to have meaningful conclusions [34].

Under this definition of workloads, they are a mandatory part so that we can control the experimentation. With a deterministic description of the usage of an HPC platform, we can vary single factors to see their relative impact. With this in mind, we can further classify them in two groups: *static* and *dynamic*. Their difference is that the first are workloads in which every job is submitted *at the same time*, abstracting away from dealing with inter-arrival time. This type of workloads is always synthetic. Meanwhile, in the former, the submitted time is also recorded. Feitelson on his web page [21] provides both real and synthetic workloads.

Workload modelling is important because it allows us to reproduce the same execution and just make single changes to measure the impact. But, it can grow considerably in its intricacy. There are many possible scenarios that can happen: like power outages, user cancelling, nodes going offline, weekends, or holidays which impact how users submit jobs, and – consequently – how the machine is used.

2.5.1 Static Workloads

The main idea of static workloads is to take a snapshot, instantaneous, moment of a particular state of the machine. Thus, in this case, it is only necessary to have a description of the resources requested for each job and the user who launched it. This technique is employed, for example, in the work by Jeannot et al. [9] to abstract away from the inter-arrival times and provide a distressed scenario for their technique.

The drawback of using static workloads, is that, compared with the dynamic workload, they have a shorter span under which the simulated scenario resembles reality. At some point in the simulation, we have that it will monotonically decrease the usage of the machine.

2.5.2 Dynamic Workloads

A dynamic workload is one in which the inter-arrival time of jobs plays a role. This is the natural type of workloads. Besides the job resources requested information, we require the submission instant. As mentioned, dynamic workloads can be *real* of *synthetic*.

Modelling accurate synthetic dynamic workloads is tricky since there are various patterns of self correlation, positive and negative feedbacks, cyclic patterns, and many other behaviors to account for. For example, in image 2.4 taken from [21] we see how the submission, or arrival, of jobs or total CPUs requested vary considerably between day of the week and time of the day. In the book by Feitelson [18] there is an in depth analysis of the behaviors.

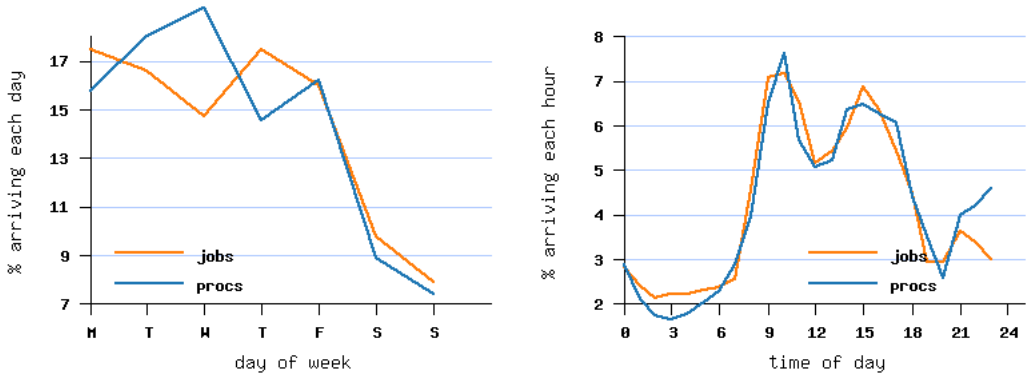

Figure 2.4: Week variation of total CPUs used and number of jobs submitted in CEA-Curie. Plot taken from Feitelson's Workloads Archive [21].

2.6 Computational Workflows

A *computational workflow* is a description of a process with dependencies, i.e. a job or a task has to wait until all of its predecessors finish, under which data flows in between them. This adds a layer of abstraction that separates the domain specialists of the computing specialist. The former is charge of what is computed, while the latter on how to run it. Conceptually, a workflow is a set of tasks with an associated partial ordering. This abstraction is the building block for the organization of workflows applied in many different domains. To name examples in each domain: engineering [2], biology [3], Earth sciences [1], and business [35].

Complex Earth Science simulations are a subset of *Computational Workflows*. Examples in Earth Science simulations, are workflows of the climated models such as EC-Earth [36], ICON [37], IFS-NEMO [38], both in weather forecast simulations and climate simulations. Within the Earth Science department of the BSC, the associated workflow of a model is usually the model name prepended of "auto". Then, for the EC-Earth model, the workflow is auto-EC-Earth, for example.

Additionally, under resource sharing platforms, workflows are necessary to allow the execution of lengthy simulations. Because of the resource sharing, resources are hard bounded both in runtime and allocated CPUs. This obliges the users to split the simulation in chunks, which are the simulated time segments of the simulation to make the execution time be lower than the wallclock of the machine. Moreover, depending on how the simulation is set, there are multiple steps that need to be orchestrated to receive the data generated by the models – e.g. post processing tasks such as data standardization or statistics –.

In order to manage and execute workflows, *workflow managers* or *meta schedulers* were developed [39] [40] [41] [42] [43]. These tools automate every step of the workflow which composed of: submission for execution, syncing of data, and retrieval of data. The workflow itself is described in human-readable form, which is then parsed and turned into tasks in the workflow engine.

Normally, computational experts develop and maintain these descriptions for each application, known internally in the Earth Science department as auto modellers. They are tasked with understanding the data flow of the workflow and the necessary software and hardware dependencies of each one of the tasks to run in each platform.

With the auto model built, this is handed to the final users, which are scientists and operationals, who are those in charge of understanding the myriad of parameters which control the physical behavior of the simulation and interpret the output.

In this work we have used one of the auto models developed in the Earth Science department as a reference: auto-MONARCH [44], which is build on top of MONARCH [1]. We use the configuration that was used in the validated reanalysis of 2023 for the CAMS project. This step is to rerun the model with the observed data from last year, 2022, so that the physical parameters can be corrected so that the model agrees with the observational data. Its most demanding task is the dynamical core which running in 96 CPUs, or 2 nodes, it lasts about 30 minutes.

2.7 Autosubmit

Out of all *workflow managers* or *meta schedulers*, Autosubmit [39] differentiates itself as being tailored for running climate simulations and weather forecast simulations. Modern weather forecast simulations, climate simulations, and air quality simulations are made up of multiple models and could be required to run in multiple HPC platforms, with various configuration steps and data management. This is a far cry from just 10 years ago when most scientists had only to work on a single,unrestricted, platform and a couple of models. This motivated the creation of the Autosubmit tool, which automatically deals with the submission of jobs, respecting the dependencies, to the various platforms whilst keeping track of the data provenance.

Autosubmit is a Python tool to create, submit, and monitor experiments. With just a few changes on the configuration file, it is able to change the HPC platform, the scheduling configuration, and the workflow. It also abstracts away from the scheduler, allowing it to support various: Slurm [10], LSF [45], PBS [46], SGE [47], and in the future, PJM [48]. As HPC platforms might fail, and often do, Autosubmit also provides

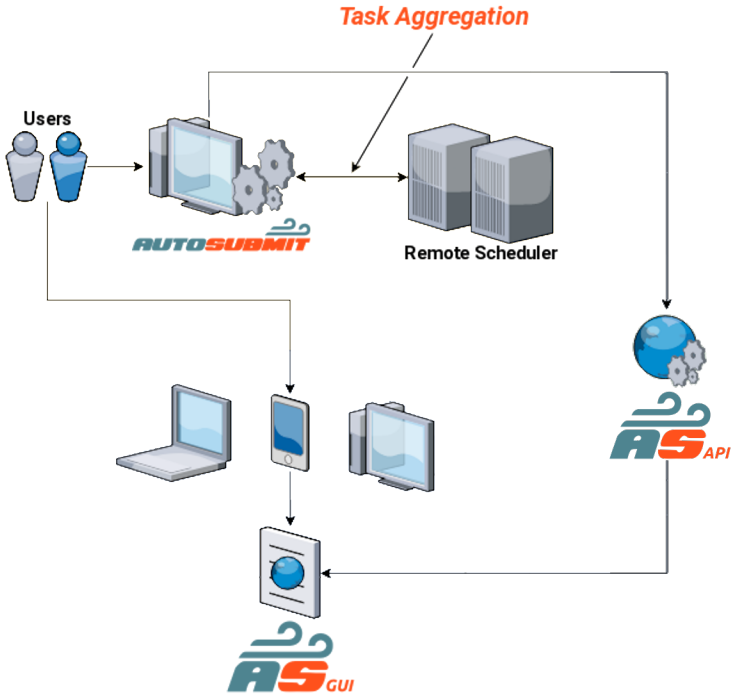

Figure 2.5: Diagram of the components of Autosubmit and its interfaces.

robust fault tolerance. In figure 2.5 we observe the different interactions that Autosubmit has: starting with the users that execute it, the two way communication it has with the remote scheduler, and the web graphical interface.

This software is currently used at the BSC to run state-of-the-art models – like EC-Earth [36], MONARCH [1], and NEMO-IFS [38] –, operational toolchains – S2S4E [49] –, data-download workflows – ECMWF's MARS [50] –, audio processing pipe line – Language Technologies group at the BSC –. These workflows are run using Autosubmit in many different supercomputers: MareNostrum 4 [11], LUMI, Levante [13], Meluxina [14], Cirrus [51]. Moreover, Autosubmit will be used in leading flagship European research projects: Earth Digital Twins [52], Digital Twin Oceans [53], and Eddy Rich Earth System Models [54].

2.8 Wrappers

To tailor the task submission in order to optimize queue time, Autosubmit developers came up with a technique called wrappers. This tackles the two way interaction between Autosubmit and the remote scheduler, seen on figure 2.5. The idea is to create a single larger job which comprises several tasks respecting restrictions as defined by the workflow.

There are two elemental types of wrappers: vertical and horizontal. The first creates a script where jobs are sequentially executed, the latter runs in parallel the tasks. In both cases, this script is submitted a single time to the HPC platform. In figure 2.6, we see inside the red box the jobs that are submitted in a single package to the HPC platform. Their dependencies are indicated by an arrow connecting two boxes, which are different jobs. As for the horizontal wrapper, we see in figure 2.7 – again inside a red

box – the jobs that are packaged into a single larger one. Here there are no dependencies among them, so it is possible to run them concurrently.

Additionally, there is also a mix-and-match of elemental types: vertical-horizontal and horizontal-vertical. The first one is made up of multiple vertical wrappers joined in a single horizontal. Analogously, the latter is build from various horizontal jobs that run sequentially in lock-step. So, for example in figure 2.8, inside the red box, jobs could either be executed one row at each time or each list runs concurrently respecting the dependency of the jobs.

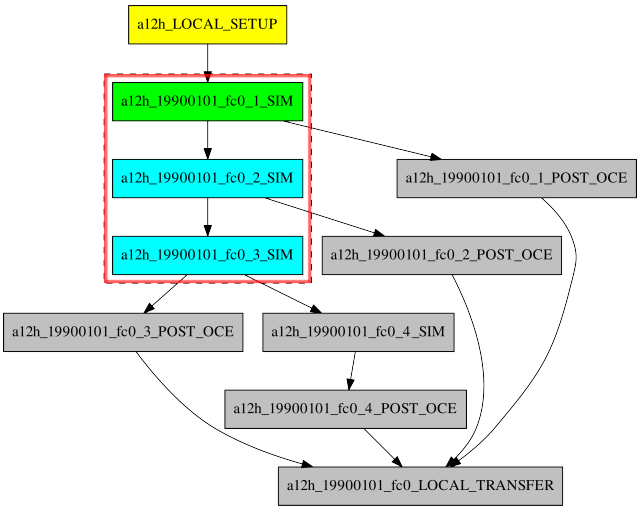

Figure 2.6: Vertical wrapper workflow example. Here the three SIM jobs will be submitted to platform at the same time, and run sequentially.

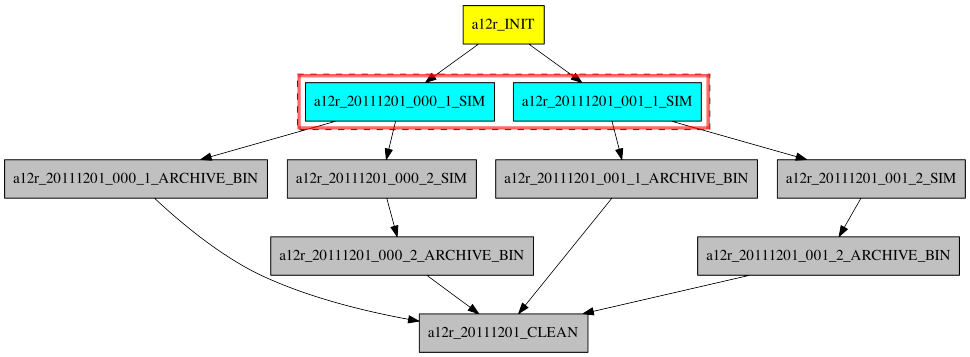

Figure 2.7: Horizontal wrapper workflow example. Here both SIM type jobs will be submitted at the same time and run concurrently.

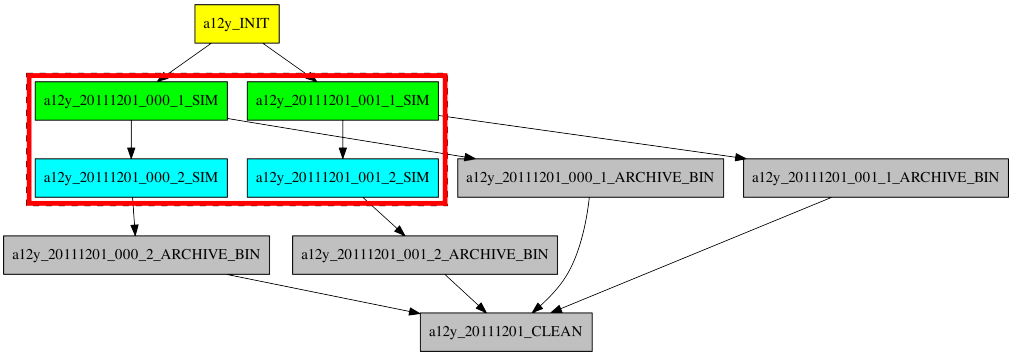

Figure 2.8: Horizontal-vertical or vertical-horizontal wrapper workflow example.

Chapter 3

Methods

In this chapter we will address the software that we used, developed, and adapted to carry out the simulation. Moreover, we also describe how we gathered and analyzed data to build a synthetic static workload that follows LUMI's supercomputer job size and runtime distribution.

3.1 Software Stack

In this section, we will give an overview of the software tools that we used to carry out simulations and the tools that were developed.

3.1.1 Slurm simulator

We used BSC's Slurm simulator [23], which is an open source software for simulating the popular Slurm scheduler. It is an enhanced version of UBCCR's simulator [24]. The original authors aimed to have a simulator that ran mostly the code of Slurm but it is able to read from a workload description file, like the Standard Workflow Format [20], and obviously run faster than real life execution – e.g. 1 month of simulated time in 1 day –. Since it runs on Slurm code it allows us to configure the scheduler as in real life platforms for the purpose of this work.

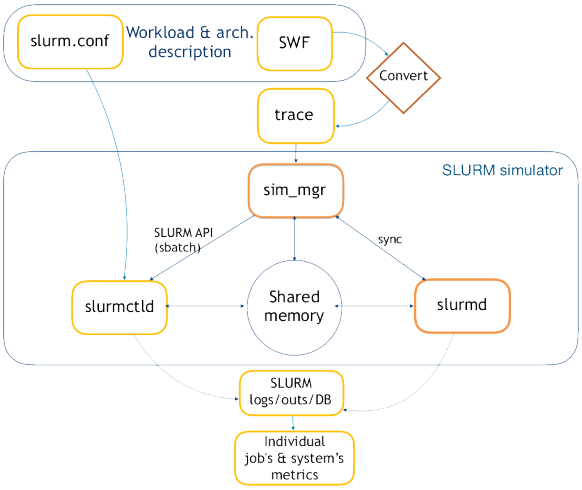

Figure 3.1: Diagram of the Slurm simulator. Taken from [23].

The Slurm simulator, as is Slurm, is composed of several executables – as explained in

section 2.2 – that have to be properly configured and orderly launched. For this purpose the developers of the simulator provided a launch script. We translate to Python this script and extended it[1] to also launch an additional daemon: the *slurmdbd*, which is in charge of the user and system accounting done with SQL databases. On top of tracking usage, it is the only way of configuring users under accounts.

In order to describe the workload to be simulated on an artificial machine, we use the Standard Workflow Format, which was proposed by Chapin et al. [20]. With it, we can determine the users and the accounts under which they execute the tasks. We will make a simplifying hypothesis and *only consider 1 partition and 1 cluster*, then the tuple of the association only changes for the user and the account. Upon including every association, using the *sacctmgr* utility, which is the Slurm account manager tool, we need to add a *RawShare* value for each user. We assign every association – i.e. user and account tuple – the same *RawShare*, 100.

Once the users are set, we proceed to convert the workload file from SWF to a format accepted by the simulator, which is binary. The developers of the BSC's simulator provided a conversion tool from SWF to binary. With that set, we can launch the simulator manager, which is the algorithm in charge of actually reading the jobs and submitting them to the *slurmctld* controller via *sbatch*.

Since user usage impacts the fair share value, as seen in equation 2.4, it is important for us to ensure that the simulation runs on a sanitized environment. Docker [25] provides such capability, on top of being light weight compared with using a virtual machine. Hence, we developed a Docker image[2] for the Slurm simulator. We design the image so that the simulator only needs two binded files – that is, synced files between the host system and the container – an input SWF and an output file. Optionally, we can bind different configuration files and the logs that are produced by the simulator. This made changing configuration files easier and also allowed for debugging the simulator. Upon converting the SWF file to binary, the allocated resources are rounded up to next number of CPUs as multiple of the total number of CPUs in a node – e.g. in a system where nodes have 64 CPUs, a job requesting 129 CPUs would be converted to a job requesting 192 CPUs –. This is due to a limitation that the simulator has that it can only run jobs as if they were requesting *exclusive nodes*. As a corollary of this property, it is not possible to allocate less than 1 node for a job using the Slurm simulator.

3.1.2 Standard Workload Format tool

All the manipulation of SWF [20] files is done via a tool developed within this work in Python, called *PySWF*[3]. This tool is used to build the workload files, with the workflow, and then to compute the associations to be sent to the *sacctmgr* utility. Besides that, we can read from multiple file type outputs: from SWF, *sacct* output with parseable flag, and from the Slurm simulator output format. And, afterwards, it can write in SWF file.

The job list is read and stored in a Pandas [55] Data Frame [56] object, which is a highly versatile, easy to manipulate, feature-full data structure with ample statistical analysis capabilities. Moreover, it also provides plenty of ways of outputting the data: CSV, FWF, etc.

Additionally, since we also support reading from real-life logs, we provided an anonymization feature, using Panda's factorize method, which maps all the different values to integers. Then to anonymize, we forget this map. This was used for the UID, account,

[1] Available at `https://earth.bsc.es/gitlab/mgimenez/ces_slurm_simulator_tools`.
[2] Available at `https://earth.bsc.es/gitlab/mgimenez/docker-ubuntu-ces-slurm-sim`.
[3] Available at `https://earth.bsc.es/gitlab/mgimenez/pyswf`.

Factor	Weight
Fair share	100000
Age	100000
Job size	10000

Table 3.2: Weights of the MareNostrum 4 for the multifactor priority computation, as seen in equation 2.4.

Option	Value
default_queue_depth	10000
defer	True
sched_interval	60 *s*
bf_interval	60 *s*
bf_max_time	30 *s*
bf_resolution	1800 *s*
bf_window	10080 *min*
bf_continue	True
PriorityMaxAge	10 days

Table 3.3: Main and secondary scheduler configuration, as seen in figure 2.2. These are used in MareNostrum 4 and we use them in all of our experiments.

partition, and QoS.

3.2 Slurm configuration

In table 3.1 we break down the differences among the two Slurm configuration files that we use in this work. On both of them we set the priority to be computed using *multifactors*, covered in section 2.4, and compute the fair share with the *Fair Tree* algorithm, seen in subsection 2.4. On tables 3.2 and 3.3 we have the weights for the priority and the scheduler options used in MareNostrum 4's configuration – as of June 2023.

MareNostrum 4's scheduling policy is used because it is a shared resource platform, which is highly used as seen in image 1.1, and it is a modern system – although it is at the end of its life cycle. Moreover, is a machine which we had access to its Slurm configuration file.

	Simulated Machine	Physical cores	Secondary Scheduler
Configuration file 1	LUMI	360448	Backfill
Configuration file 2	CEA-Curie	93312	Backfill

Table 3.1: Description of the Slurm configuration file we use in this work.

For the user distribution, we did another simplifying assumption and consider only a 1 level tree, as seen in figure 3.2. This is done to make it more convenient the computation of the fair share factor, which we will need in order to control the fair share of the user that launches the workflow. Below the root account, we have a full row of accounts. Every account has at least 1 user, while the user that we track his job is in a separate account. The associations are taken from the SWF file. We also consider that every user could have at most 1 account associated with it.

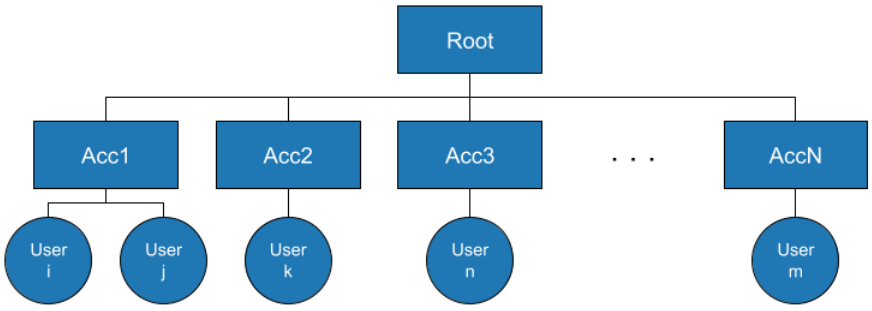

Figure 3.2: Diagram of the user structure we consider for the synthetic and dynamic workloads in this work.

3.3 Workloads

We use two types of workloads: static and dynamic. For the first, we *generate the workload* from the analysis of data from the LUMI supercomputer, while the second was taken from Feitelson's repository of workloads [21]. Hence the first is a synthetic workload whilst the second is a real one.

3.3.1 Static

With static workloads, we aim to study the relative impact of individual job attributes and user attributes with the wait time in queue. We model the static workloads by fitting appropriate distributions to the runtime and to the allocated CPUs as we have observed in jobs from the LUMI supercomputer. This computer's data is used because it is a new system [12], its resources are highly sought after, and it is meant for scientific purposes.

Static workloads gives us a short-sighted view of the evolution of the system. But it allows us to abstract away from the inter-arrival time and reduce somehow the "what if"s that this area has.

Job Gathering

The historical data in LUMI was captured by performing *sacct*, Slurm account, commands which retrieve job log information from Slurm's databases. We are interested in the total allocated CPUs, submit, start, UID, QoS, and partition. Also we fetched the User id of the user that launched the job. With this information we have more than 700000 jobs from LUMI, which can be seen in table 3.4. An important note is that LUMI's CPUs support hyperthreading and jobs take advantage of that feature *by default*. This means that some jobs are reporting their usage in logical cores, whilst others are reporting it in physical cores – if the user has explicitly removed hyperthreading.

Job Analysis

In this section we build a description jobs run on the LUMI supercomputer. We followed closely Feitelson's book on modelling [18]. Since we are interested in building a static workload, i.e. we do not care about the relative arrival time in between jobs, we focus only in *two attributes* of the job: runtime and allocated CPUs. To read the data, we used PySWF module 3.1.2 that reads the data as outputted by the *sacct* Slurm function, with

Cluster	LUMI
Num of jobs	$\geq$ 700000
Num of users	$\approx$ 800
Start date	Feb 2023
End date	Jun 2023
Total CPU nodes	1536
Total GPU nodes	2460
Total Physical CPUs	354048
Total Logical CPUs	720896

Table 3.4: Characteristics of the LUMI supercomputer as of June 2023 and also the dataset that we have collected.

parseable flag. With the data read, we used Scipy's [57] statistical module for analysis features to represent data and to fit distributions by maximizing log likelihood.

We start by analysing the data by computing its mean, standard deviation, and quartiles. In table 3.5, there is a large difference between the median and the mean, indicating a skewed distribution. This has been noticed by other authors [18] [17] for the allocated CPUs, where a vast majority of jobs are small but there are some super large jobs, which causes the mean to increase. For example, in our data we saw a single job request of 325120 CPUs, while half of them require less than 256 CPUs.

As for the job runtime, in table 3.6, we observe the same large difference between the mean and the median as in the allocated CPUs table 3.5.

Consequently, if both runtime and allocated CPUs, tables 3.5 and 3.6, have this skewed property, the multiplication of both is also expected to have the same behavior. In table 3.7 we compute the core runtime in $CPU \cdot s$, and indeed observe what we expect.

Additionally, we plot a log allocated CPU per log runtime scatter plot of the jobs, in figure 3.5. We observe equidistant lines, which are the power of 2 jobs. In terms of runtime we don't see much structure except for the 2^{17} horizontal line, about 172800 seconds or 2 days, which is the maximum wallclock allowed in the partitions which all users have access, *standard* and *standard-g*.

We fit a skewed distribution onto the runtime and the allocated CPUs. Hence, we choose 4 distributions to fit which have appropriate characteristics: positive support, long tailed distributions because we know that the behavior on HPC platforms follow this trend [18] [19] [17]. Moreover, as we can see in the summarized statistics tables 3.5, 3.6, and 3.7 the difference between the mean and the median, 50%, is considerable. This indicates us the long tail behavior. The distributions' probability function we fit are:

1. Lognormal:
$$f(x;s) = \frac{1}{xs\sqrt{2\pi}} e^{-\frac{ln^2 x}{2s^2}}, \quad x > 0 \tag{3.1}$$

2. Log-uniform:
$$f(x;a,b) = \frac{1}{x \cdot ln\frac{b}{a}}, \quad a < x < b \tag{3.2}$$

3. Weibull (or Weibull minimum, in Scipy's nomenclature):
$$f(x;c) = cx^{c-1}e^{-cx}, \quad x \geq 0 \tag{3.3}$$

4. Pareto:
$$f(x;b) = \frac{b}{x^{b+1}}, \quad x \geq b \tag{3.4}$$

Statistic	Value
Mean	774.2
Std	4493.4
Min	1
25%	48
50%	256
75%	1024
Max	325120

Table 3.5: Number of allocated CPUs statistics on LUMI. 25% refers to the first quartile, 50%, the median, and 75%, the third quartile.

Statistic	Value (s)
Mean	3952.7
Std	19617.9
Min	0
25%	23
50%	125
75%	902
Max	4604100

Table 3.6: Job runtime statistics in seconds on LUMI. 25% refers to the first quartile, 50%, the median, and 75%, the third quartile.

In order to provide a notion of fitting that takes into account over fitting, we also computed the Akaike Information Criterion [58], besides the Sum of Squared Errors. The AIC is computed with the following formula:

$$AIC = 2k - 2log(\hat{L}), \tag{3.5}$$

where k is the number of parameters in the model, and $\hat{L}$ is the maximized log likelihood value of the observed values with respect to the fitted distribution.

Scipy uses a standardized way of representing distributions, with two parameters: *loc* and *scale*, which are always present in the description of the distribution. But some distributions require additional parameters to describe them, which we include as additional columns in tables 3.10 and 3.11. We also plot the cumulative distribution of the log transform in plots 3.3 and 3.4. In the first, we observe that about 60% of the jobs have less than $e^6 \approx 403$ CPUs allocated. Similarly, for the latter, we see that about 50% of the jobs with runtime different that 0 run for less than $e^5 \approx 148s$ seconds.

Statistic	Value ($CPU \cdot s$)
Mean	3.420701e+06
Std	1.392048e+08
Min	0
25%	3.533000e+03
50%	3.942400e+04
75%	2.150400e+05
Max	9.505604e+10

Table 3.7: Job core runtime in $CPU \cdot s$ statistics on LUMI. 25% refers to the first quartile, 50%, the median, and 75%, the third quartile.

model	SSE	AIC	Δ SSE	Δ AIC
lognorm	1.5212695e-07	-10982364.7	3.490224e-08	8.099168e+05
loguniform	2.4445086e-07	-11792281.5	1.272262e-07	0
weibull_min	1.1722471e-07	-10953333.3	0	8.389482e+05
pareto	1.242175e-07	-10972152.5	6.992826e-09	8.201290e+05

Table 3.8: Distribution fitness for the allocated CPUs. AIC is computed as 3.5. Δ fields are with respect to the best SSE or AIC, which is the minimum.

model	SSE	AIC	Δ SSE	Δ AIC
lognorm	1.4533082e-09	-10949897.1	1.582374e-10	5.908307e+05
loguniform	1.2950708e-09	-11540727.9	0	0
weibull_min	1.3592729e-09	-10750257.9	6.420204e-11	7.904700e+05
pareto	1.5197229e-09	-10990301.6	2.246521e-10	5.504263e+05

Table 3.9: Distribution fitness of the runtime for the runtime of jobs in LUMI. AIC is computed as in equation 3.5. Δ fields are with respect to the best SSE or AIC, which is the minimum.

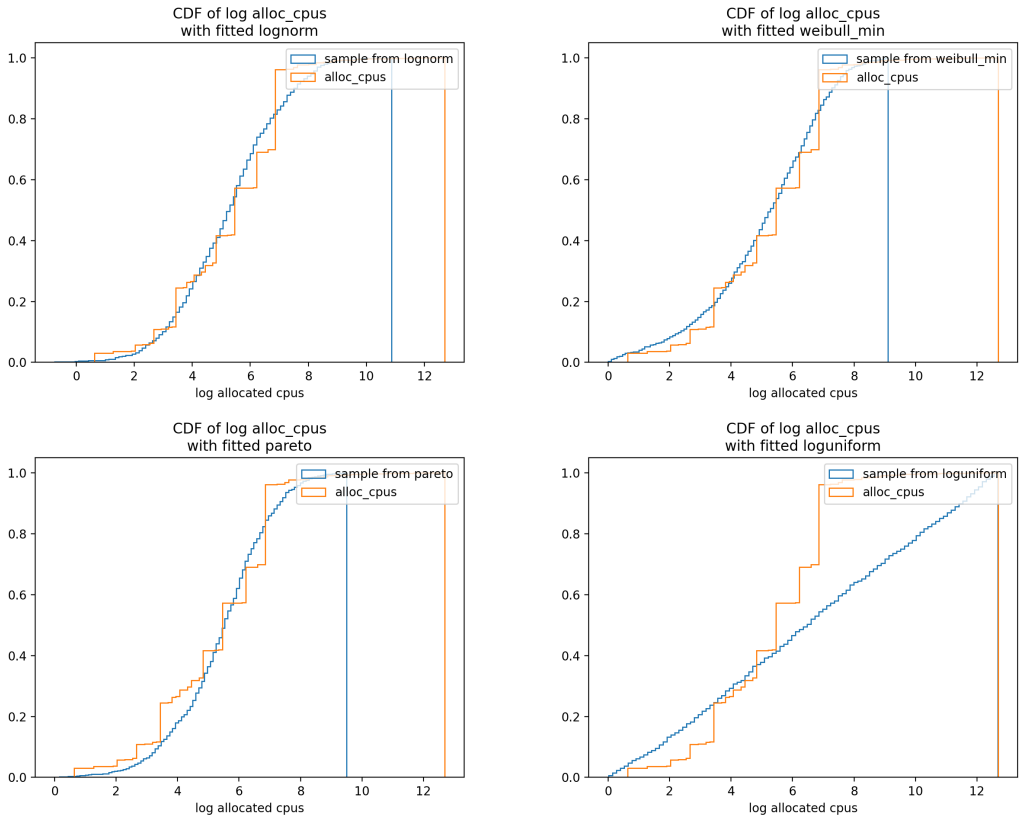

Figure 3.3: Cumulative distribution function of the fitted distributions for the allocated CPUs at LUMI. The orange line is the log allocated CPUs observed cumulative distribution and the blue line is a random sample generated with the parameters as found in table 3.10.

model	loc	scale	s	a	b	c
lognorm	-9.8930e-1	1.9930e+2	1.7195	-	-	-
loguniform	-5.0045e-1	1.0000	-	1.5004	3.2512e+5	-
weibull_min	1.0	4.2983e+2	-	-	-	0.60622
pareto	-4.4421e+2	4.4521e+2	-	-	1.6045	-

Table 3.10: Fitted parameters of the distributions for allocated CPUs. *Loc* and *scale* reference Scipy's [57] standardization of the distribution. s, a, b, and c are as in equations 3.1, 3.2, 3.3, and 3.4.

model	loc	scale	s	a	b	c
lognorm	9.5007e-1	1.6506e+2	2.6766	-	-	-
loguniform	-7.6965e-1	1.0000	-	1.7626	4.6041e+6	-
weibull_min	1.0	6.4232e+2	-	-	-	3.8084e-1
pareto	-3.2248e+1	3.3248e+1	-	-	4.4822e-1	-

Table 3.11: Fitted parameters of the distributions for runtime. Loc and scale reference Scipy's [57] standardization of the distribution. s, a, b, and c are as in equations 3.1, 3.2, 3.3, and 3.4.

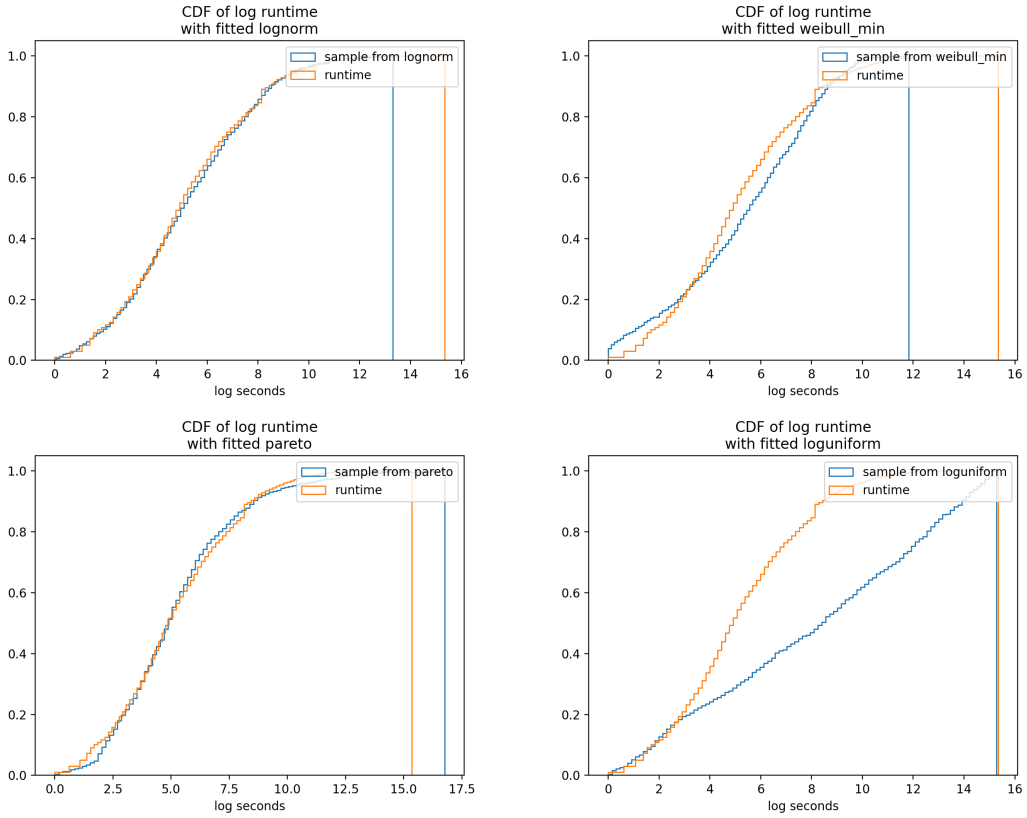

Figure 3.4: Cumulative distribution function of fitted distributions for the runtime at LUMI. The orange line is the cumulative log runtime of jobs with runtime different than 0 and the blue line is the cumulative distribution of a random sample generated with the parameters as found in table 3.11.

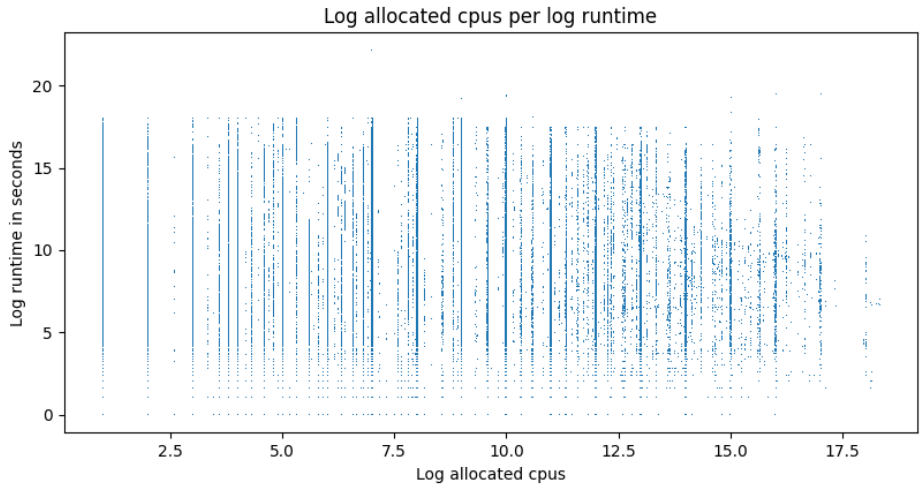

Figure 3.5: Log runtime per log allocated CPUs. Jobs with runtime 0 were discarded. Each pixel represents a job. This plot was zoomed to the area of highest concentration of jobs.

Workload Generator

In this section we will detail how we generate a workload that follows the distribution for both the runtime and allocated CPUs that we observe on the data we collected.

With the appropriate minimized parameters, we used Scipy's methods to generate values according to the fitted distribution. All the considered distributions were positive, but not necessarily bounded. Then, we need to take a step to bound them to the maximum number of CPUs in the system, without losing the property of being a probability distribution. Following [18] we do the inverse transform method which ensures us that we still have a probability distribution. Finally, the generated values are casted to integer, since all the distributions that we consider in this work are continuous.

Upon analyzing tables 3.4 and 3.9 and visually inspecting 3.3 and 3.4 we choose to use the lognormal distribution for both the runtime and allocated CPUs. And by the log-log plots we observe that the log normal provides a better fit with respect to the shape of the cumulative distribution that the log-normal, that is, smaller jobs are better represented by the distribution.

Once we generate a job attribute characteristic: runtime and allocated CPUs, we attribute it to a user chosen *uniformly at random* from a population of 100 users. Since we also need to bundle users onto groups, if a task is attributed to a user without a group, we *uniformly at random* assign this user to a group from a population of 100 as well.

We generate workloads with 1000 jobs, which we choose because it provides a distressed enough scenario, under which we have queues forming. Also, this number of queuing jobs is common, as seen in MareNostrum 4 plot of queuing and running jobs 1.1.

In tables 3.12, 3.13, and 3.14 we have the descriptive analysis of the aggregated generated workload. Moreover, in tables 3.15 and 3.16 we have the proportion for both number of jobs and usage distributed per users for the most relevant ones.

Statistic	Value
Mean	959.9
Std	4599.1
Min	1
25%	63
50%	199
75%	650
Max	253375

Table 3.12: Job allocated CPUs statistics on a synthetic static workload generated from LUMI. 25% refers to the first quartile, 50% the median, and 75% the third quartile. Aggregated across all 10 generated files.

Statistic	Value (s)
Mean	2735.8
Std	11194.4
Min	0.00
25%	25.00
50%	155.00
75%	940.25
Max	162234

Table 3.13: Job runtime statistics on a synthetic static workload generated from LUMI. 25% refers to the first quartile, 50% the median, and 75% the third quartile. Aggregated across all 10 generated files.

Statistic	Value ($CPU \cdot s$)
Mean	2.234767e+06
Std	2.826463e+07
Min	0.000000e+00
25%	3.817500e+03
50%	2.990450e+04
75%	2.643082e+05
Max	2.337595e+09

Table 3.14: Job core runtime ($CPU \cdot s$) statistics on a synthetic static workload generated from LUMI. 25% refers to the first quartile, 50% the median, and 75% the third quartile. Aggregated across all 10 generated files.

uid	Total Core runtime (%)
33	50.76
59	8.10
57	7.80
37	4.97
7	4.42
98	2.60
31	1.86
32	1.77
39	1.72
48	1.34
78	1.00

Table 3.15: Job core runtime proportion in percentage per user in one of the synthetic workload generated based on LUMI. Users with less than 1% of the total runtime, 46060155.26 $CPU \cdot s$, of the workload were omitted.

uid	Total jobs (%)
31	1.90
64	1.80
30	1.70
82	1.70
78	1.60
72	1.50
4	1.50
86	1.50
76	1.50

Table 3.16: Total number of jobs per user in the seventh synthetic workload based on LUMI. Users with less than 1.4% of the number of jobs, 14, of the workload were omitted.

Experimental Design

In this section, we specify how we set up the experiments: how we combine job configurations with different fair share factors and measure their impact.

As mentioned in section 3.3.1, we generate 10 synthetic static workloads. Into each of these, we include a single job with every combination of 30, 60, 120, and 210 minutes of runtime and 96, 192, 384, 672, and 1152 CPUs. We used the CAMS workflow dynamical core job as a reference for this values, as we have mentioned in section 2.6. Usually, for the CAMS use case in the Earth Science department this job takes 30 minutes when run with 96 CPUs.

The rest of the cases are multiples of it, as if it was wrapped vertically, horizontally, or both. For example, the 96 CPUs with runtime 120 minutes is if it was 3 vertical jobs, or 384 CPUs and 210 minutes if it was a horizontal-vertical wrapper with 4 sequential jobs and 3 parallel jobs. We will call the user that launches this controlled job the *workflow executing user*.

Since there are no executing prior to the instant where all the jobs are launched, every user would have fair share of 1.0. We want to resemble an instant of the typical workload of a share platform machine, then we added to the generated workload a first batch of jobs so that users have *usage recorded by the Slurm simulator*. This first dummy

batch of jobs, one for each user, only differ in runtime. The number of allocated CPUs was chosen so that there were enough resources in the machine for *all the jobs* in this first submission.

However, we still needed to actually decide which user would have more or less usage. We set the runtime so that the relative usage after the execution of this dummy job in between users is preserved with respect to the generated usage – that is, the user with the highest usage will have the highest runtime of all the dummy jobs; the second highest user will have the second highest runtime and so on and so forth – as seen in figure 3.6 –. Except the *workflow executing user*, which we change the runtime of his dummy job to according to the fair share we want to set to him. We opted to preserve the relative order - after traversing the Fair Tree algorithm seen in section 2.4 - of the rest of the generated users because we know that users tend to have recurrent patterns of usage as reported by [18] and [26].

So, on top of all the combinations of job attributes, we added one more dimension which was a different fair share value for the *workflow executing user*: 0.1, 0.2, 0.25, 0.3, 0.5, 0.6, 0.7, 0.75, 0.8, and 0.9.

With this analysis, we hope to compute the relative impact of varying each job or user attribute for just a single submission.

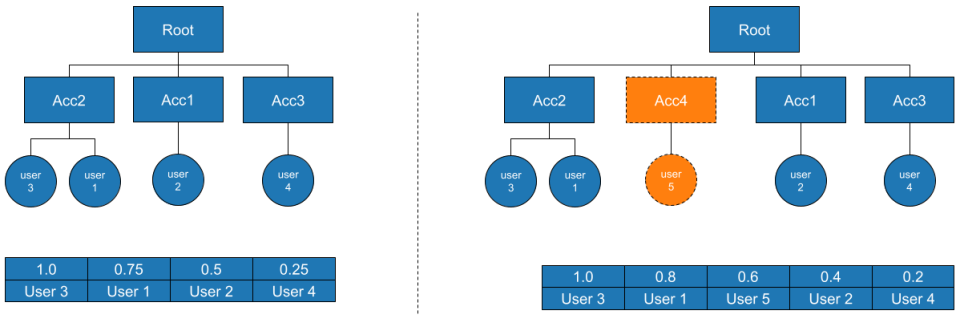

Figure 3.6: Example of the fair share control. In orange and dashed line border we see the *workflow executing user* and account, respectively the circle and the box. The left image is the traversing of the Fair Tree 2.4 as it was generated *after executing* the workload in the simulator. As we include the *workflow executing user*, we set the runtime of his dummy job so that it has a fair share of 0.6 *prior to the execution* of the generated workload.

3.3.2 Dynamic

In this section we take CEA-Curie workload from Feitelson's workload repository [21]. This machine, although decomissioned, was a scientific machine shared among many groups. Afterwards, we exaplain how we inserted a representative workflow into it.

Workload Choice

Here we used the workload from CEA-Curie supercomputer taken from Feitelson's repository [21]. This was a general purpose machine and fits the characteristics of MareNostrum 4 in terms of resource sharing and scientific usage. We used the clean version 2 from Feitelson, which only considered the second half of the log, after a big upgrade in the machine, and removed some jobs which were most certainly badly logged (i.e. running for far too long).

Like for the LUMI supercomputer, commented in section 3.3.1, we also computed the statistics in runtime, allocated CPUs, and core runtime in tables 3.6, 3.19, and 3.20. Additionally, we provided the wait time statistics as recorded by the log 3.18. We also computed the largest consuming users in terms of core runtime and also total number of jobs submitted in tables 3.22 and 3.23.

We have that two users make up for a quarter of the total usage of the machine in terms of core runtime, anonymized uids 290 and 135. In terms of total jobs launched, we don't observe a concentration of the jobs launched by a handful of users. The largest takes about 5% of the total jobs launched during the period tracked, 1st of February 2011 to 17th of October 2012.

Cluster	CEA-Curie
Num of jobs	312826
Num of users	582
Num of groups	305
Start Date	1/2/2011
End Date	17/10/2012
Total Nodes	5544
Total CPUs	93312

Table 3.17: Specification of the workload of CEA-Curie clean version 2 from Feitelson's repository [21].

Statistic	Value (s)
Mean	8001.0
Std	56053.1
Min	0
25%	0
50%	4
75%	69
Max	7610035

Table 3.18: Wait time statistics summary of CEA-Curie cluster in seconds. 25% refers to the first quartile, 50%, the median, and 75%, the third quartile.

Statistic	Value
Mean	688.3
Std	4044.5
Min	-1
25%	2
50%	32
75%	256
Max	79808

Table 3.19: Allocated CPUs statistics summary of CEA-Curie cluster. 25% refers to the first quartile, 50%, the median, and 75%, the third quartile. -1 value is of either the job was not scheduled or incomplete data.

Uid	Total core runtime (%)
290	14.37
135	12.43
366	06.68
387	06.56
235	06.10
62	05.91
295	02.06
287	02.02
61	02.01
52	01.89
288	01.50
575	01.42
39	01.40
341	01.23
459	01.14

Table 3.22: Top consuming users in terms of core runtime with usage larger than 0.01 of the total core runtime of the machine (which is 1.2839628e+10 $CPU \cdot s$) on CEA-Curie.

Statistic	Value (s)
Mean	6157.62
Std	17531.51
Min	0
25%	8
50%	124
75%	1540
Max	124615

Table 3.20: Runtime statistics summary of CEA-Curie cluster in seconds. 25% refers to the first quartile, 50%, the median, and 75%, the third quartile.

Statistic	Value ($CPU \cdot s$)
Mean	4.104399e+06
Std	4.502056e+07
Min	0
25%	83
50%	3081
75%	1.264280e+05
Max	7.987086e+09

Table 3.21: Core runtime ($CPU \cdot s$) statistics summary of CEA-Curie cluster. 25% refers to the first quartile, 50%, the median, and 75%, the third quartile.

Experimental Design

Since dynamic workloads have a longer window of execution than static, it allows us to consider more than just 1 job submission. Hence, we can simulate how real workflows behave, which are made of various jobs. We want to test the two elemental types of

Uid	Total of jobs (%)
327	5.27
305	3.43
691	3.19
387	3.16
350	2.94
487	2.23
33	2.21
270	2.07
47	1.84
24	1.62
636	1.59
584	1.50
426	1.48

Uid	Total of jobs (%)
556	1.44
290	1.35
619	1.35
609	1.29
344	1.29
549	1.28
235	1.26
518	1.26
510	1.14
375	1.10
555	1.03
423	1.02

Table 3.23: Top consuming users in total jobs with more than 0.01 of the total jobs submitted to the machine (which is 3128.26) on CEA-Curie.

wrappers in their simplest form: a vertical one with at most a single dependency and a horizontal with no dependencies, such as in figure 3.10. For the vertical case, we consider 7 sequential jobs with 96 CPUs and 30 minutes of runtime which we impose that the next one could only run after the current finishes, as seen in figure 3.7. For horizontal, we consider 12 jobs each with 96 CPUs and 30 minutes of runtime which are submitted at the same time, as seen in figure 3.8. These values are chosen because of the CAMS' dynamical core execution for validated reanalysis configuration, that with 96 CPUs takes about 30 minutes to finish. Then, the number of 7 vertical tasks and 12 horizontal task is chosen because they are within the normally used configuration of wrapped values in the department which gave us a scenario that differentiated the wrapped case from the unwrapped.

The simulator has *no support for dynamic submission times*, that is, only submit the next job once its dependency is finished. We have to specify the submission time on the workload file. Then, to minimize the waiting time of a constrained job – that is, still waiting for its dependency – of building too much queue time – and increasing its age factor –, we decide to put the submission of the vertical workflow as if they have ran in their ideal time – e.g. the second job is submitted 30 minutes after the first one, the third 60 minutes after the first one, and so on and so forth –. In image 3.7, we have in the green boxes with a plus sign inside them, the waiting time of that particular job if it was submitted as in real life Autosubmit. In orange, we have the simulator's time in queue as we include in the workload with the workflow. This image is done analogously for the horizontal workflow in figure 3.8, with the exception that we don't have dependencies among jobs, then there are all submitted at the same time.

With this experiment, we aim to test the performance of aggregation for each elemental type of wrappers, as we have seen in section 2.8. We choose the two simplest workflows to simplify the experiment, depicted in image 3.10. For each of the two types of workflows, we consider the unaggregated – or unwrapped – case, in which all the jobs are launched individually, and the aggregated – or wrapped –, where we create a *single job* grouping them all: in the case of the vertical workflow, that means a job of size 210 minutes and 96 CPUs, and in the case of horizontal, 1152 CPUs and runtime of 30 minutes.

Since not all portions of the workload yield simulated distressed enough scenarios

– that is, there were not enough jobs to have queues forming –, we do a prior analysis of the trace to find the most congested portions of the log. We finally set on the week 10/6/2012 to 17/6/2012. Then, on this week, we submit the workload on Thursday 14th and Friday 15th on 3 different times: 10, 15 and 20. We label these moments of submission on table 3.24 and, on figure 3.9, we plot those moments with respect to the evolution of in use and in queue resources. In orange we plot the total requested resources and in blue we plot the resources utilized. We observe two clear orange peaks in the daily usage cycle, indicating the daily cycle of usage, and the blue line with a hard bound in the total available CPUs, as in table 3.1. We see how it both utilization and requested resources fall down during night time. In black vertical dashed lines we have the each of the instants where the workflow will be submit.

Label	Submission Instant
A.1	14/6/2012 at 10
A.2	14/6/2012 at 15
A.3	14/6/2012 at 20
B.1	15/6/2012 at 10
B.2	15/6/2012 at 15
B.3	15/6/2012 at 20

Table 3.24: Submission instants of the workflow and their label.

Here we only test two cases for the fair share. Either the *workflow execution user* did not launch *anything* prior to the moment we chose for the workflow to start, which means that the user has the *best possible* fair share factor, 1.0, or we set a phony first job, which runs *prior to the rest of the workload* without interfering with it, in order to surpass the usage of the most consuming user until the moment the *workflow executing user* submits the first job. This means that the *workflow executing user* will have the *worst possible* fair share. In this case, it makes for a fair share factor of $6.9e-03$.

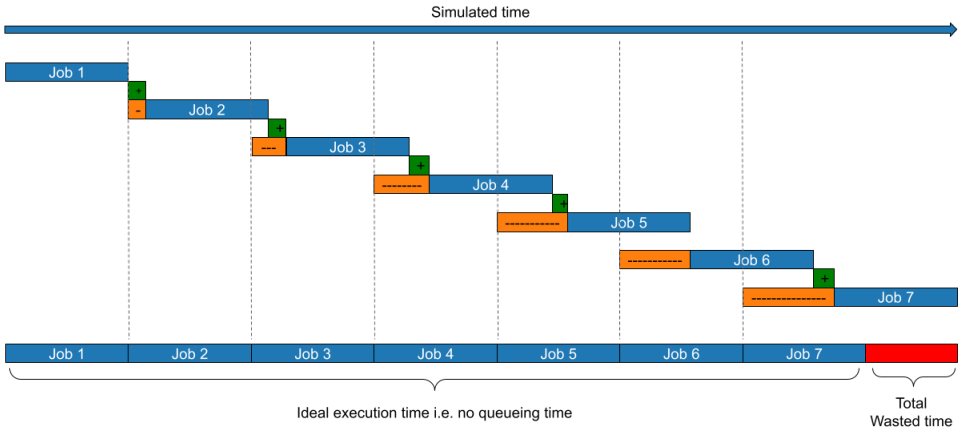

Figure 3.7: Description of the setup of the vertical workflow. The green box with a plus sign indicates the queue time if the job was launched after the end of the execution of its dependency, whilst the yellow box with a minus sign indicates the queue time as in our experiment setup.

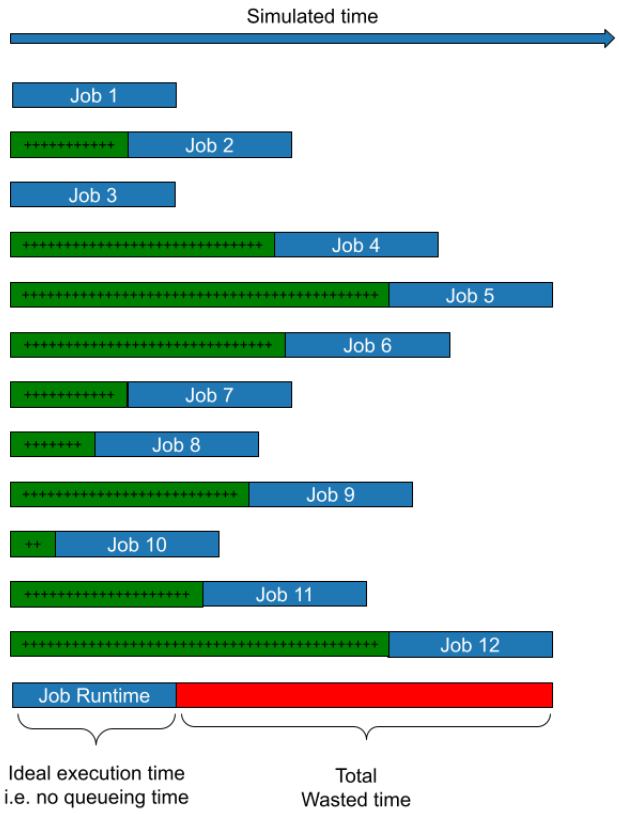

Figure 3.8: Description of the setup of the launch of the horizontal workflow. The green box with a plus sign indicates each individual job queue time.

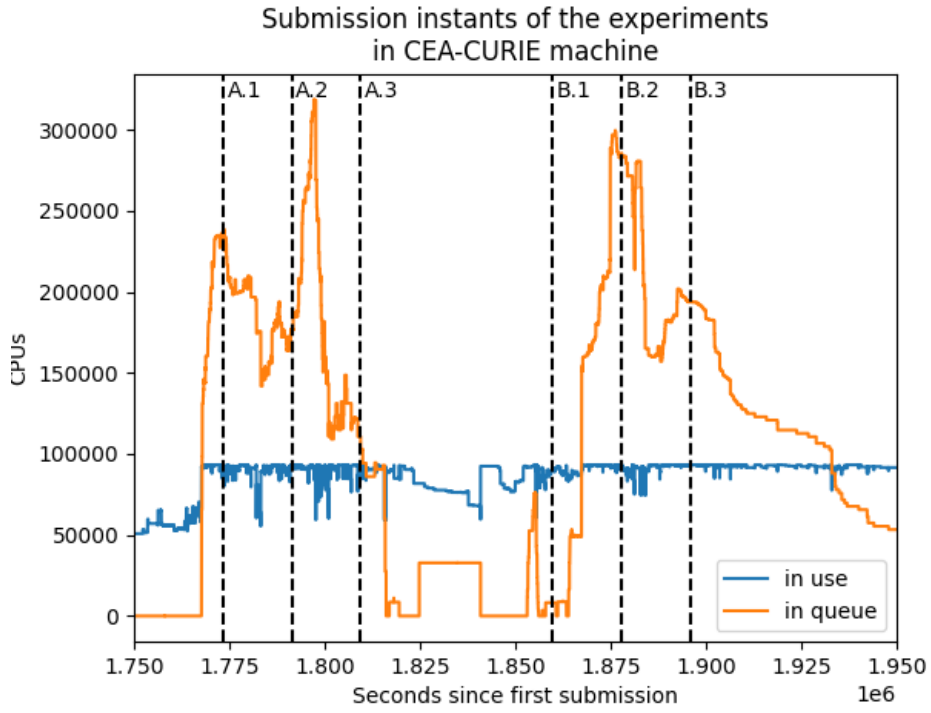

Figure 3.9: In usage and queuing resources CEA-Curie. The vertical line indicate the instant of submissions as per table 3.24.

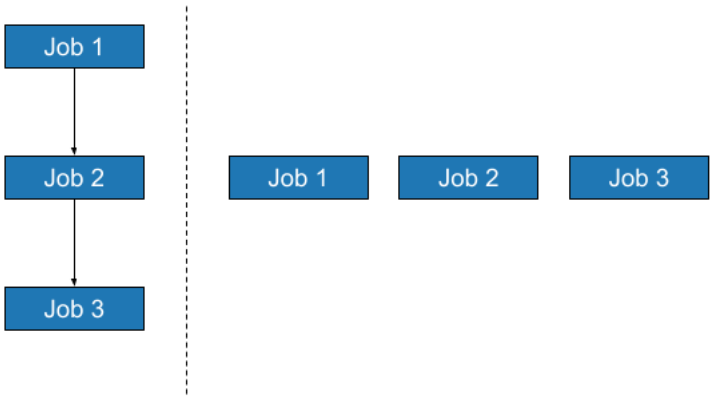

Figure 3.10: Sample with three tasks, or jobs, of the two workflows we included in the workload of CEA-Curie.

Chapter 4

Results

In this chapter we will analyse the results of the runs of both types of experiments: the static, for the impact of the job attribute, and the dynamic, to see the performance of wrappers on a long time horizon.

4.1 Job's and User's Attributes Impact

Each of the tables A.1, A.2, A.3, and A.4 are the runs with 30, 60, 120 and 210 minutes of runtime of the job launched by the *workflow executing user*. The first column is the number of CPUs allocated to the job, the following columns are the average, maximum, and minimum across the 10 experiments for both the *queue waiting time* and *priority upon start of the execution*.

We compute the correlation of the runtime, allocated CPUs with the average – indicated with a overline –, maximum, and minimum of both the queue time, T_q, and the priority upon starting, P, in table 4.1. We observe that the fair share is *the dominant factor* on the waiting time. Moreover, we see the clear inverse correlation, of -0.87, between the average wait time and fair share.

Additionally, in image 4.1 we plot the average wait time across all 10 experiments for every job configuration in function of the fair share factor. In that figure we can see that the wait time reduces in an exponential matter as we increase the fair share. For that reason, we fit an exponential model with the following expression

$$f(x; a, b) = b \cdot e^{-ax}. \tag{4.1}$$

In figure 4.3 we have the box plot of each of the configurations for a particular fair share value. This plot synthesizes the spread of the results. We see that it has about the *same spread* in between all job attributes.

For the lowest fair share factor we test, which was 0.01, we see that the average across the 10 experiments is 304 seconds with the largest time in queue of 794 and the lowest 131 seconds. With maximum queue time for the 96 CPUs and runtime 3600 seconds. For the 0.1 case, we observe a similar trend in terms of spread of across the different configurations. Although we have a 900 seconds waiting time for the second largest job in terms of core runtime: the 672 CPUs with 12600 seconds of execution time, which follows the pattern of independence of wait time with both runtime and allocated CPUs. For the subsequent fair share factors we observe an ever so increasing homogeneity, seen that every box plot looks about the same, among all the job configurations, and the span of value in the left axis decreases. Finally, all of configurations collapse onto 0 queue time from a fair share from 0.5 on wards.

We observe that the fair share is negatively correlated to the average waiting time, meaning that the larger the fair share the lower is the waiting time and vice versa. On the priority it is even more correlated, but unsurprisingly so, since jobs start executing relatively fast, hence their age factor did not grow as much, and the *fair share then dominates quickly its share of the priority*. We can check that by noticing that for 75% of the jobs the number of allocated CPUs are less than 650 in table 3.12, which adds to the priority about 18, as computed by equation 2.4, after multiplying by the weights shown in table 3.2, and the largest wait time we observe is of about $900s$ which adds 104 onto the priority. Meanwhile, a fair share of just 0.01 adds to the priority 1000.

As for the priority, we observe how the fair share is almost completely positively correlated with the average, maximum, and minimum priority. The rest of the factors, allocated CPUs and runtime show no correlation whatsoever for average, maximum, and minimum for both priority and wait time. Analogous to the wait time, we plot in figure 4.2 the average across all 10 experiments of the priority in function of fair share, for each and every combination of allocated CPUs and runtime. There, we see how the priority grows linearly with respect to the fair share, with the largest deviations in the low fair share spectrum.

Analogous to the wait time, we plot box plots for each fixed fair share value in images 4.4 and 4.5. We observe a decrease in variability as we increase the fair share. In the lower end of the values, at 0.1 notably, we observe how there are some configurations which differ its distribution from the rest: 96 CPUs and 3600 runtime and 1152 CPUs and 3600 runtime.

	$\overline{T_q}$	$max\ T_q$	$min\ T_q$	$\overline{P}$	$max\ P$	$min\ P$
Runtime	0.0048274	0.0229246	0	-0.0010371	0.0018034	-0.0001557
CPUs	0.0005572	-0.0161035	0	0.0012817	0.0062171	0.0002494
Fair share	-0.8719868	-0.8379077	-0.8572271	0.9951687	0.9911172	0.9928968

Table 4.1: Correlation table of the job and user attributes with wait time and priority. T_q is the time in queue and P is the priority upon start of the job. An overline indicates the average across the 10 experiments and max and min are the maximum and minimum observed of the particular quantity.

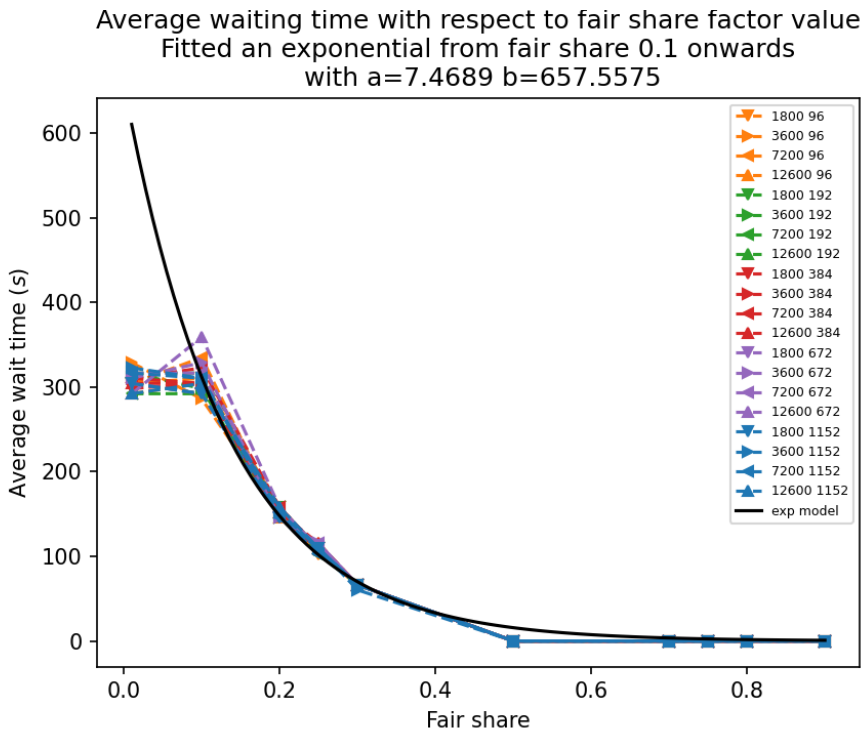

Figure 4.1: Average of the waiting time across all experiments with respect to the fair share value. Each line style is a number of CPUs allocated. Each color represents a different runtime.

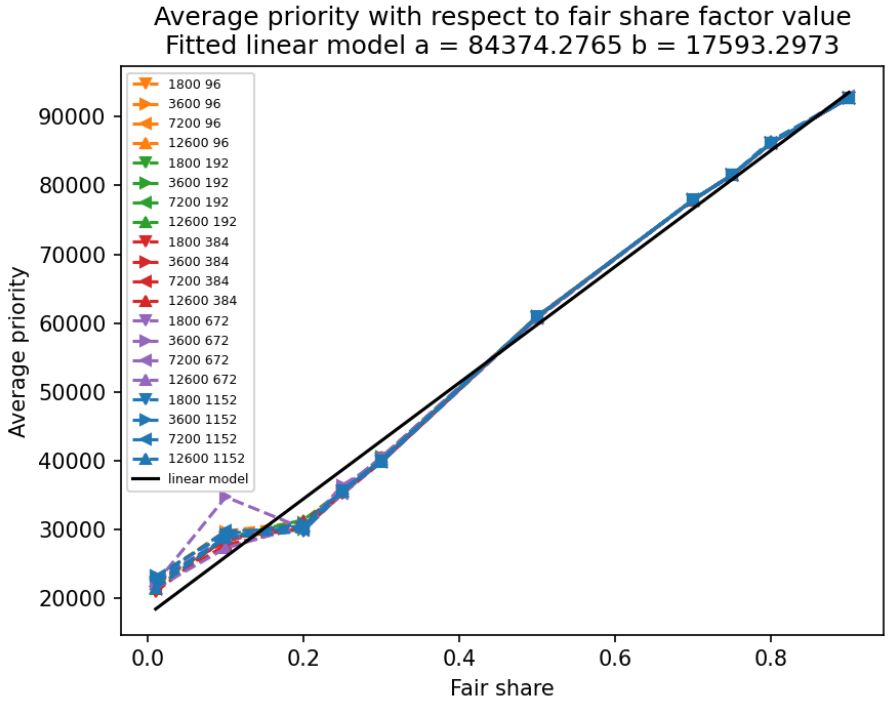

Figure 4.2: Average of the priority time across all experiments with respect to the fair share value. Each line style is a number of CPUs allocated. Each color represents a different runtime.

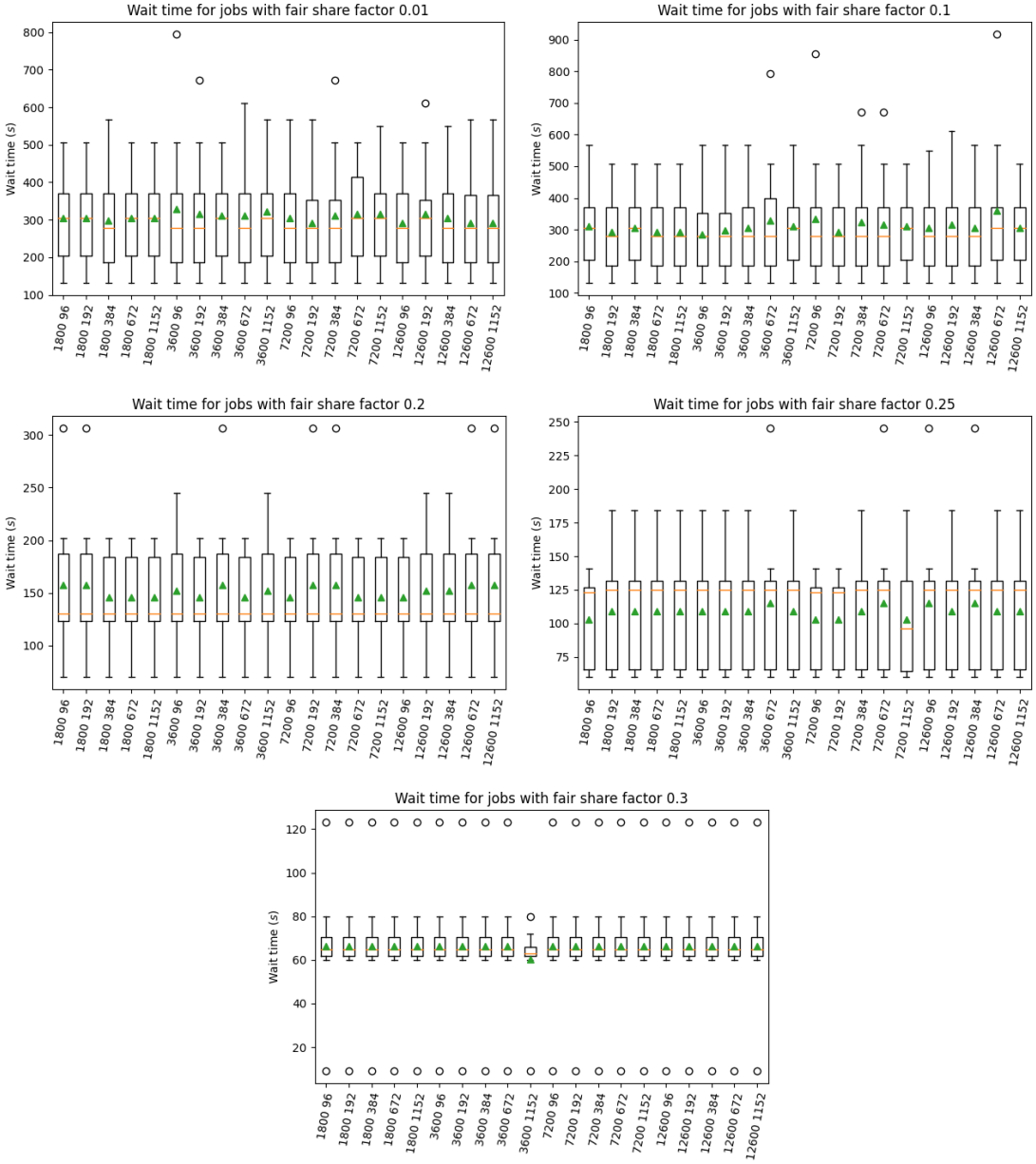

Figure 4.3: Box plot of the wait time per fair share factor. Each box plot is a combination of CPU and runtime. Green triangle indicates the arithmetic average.

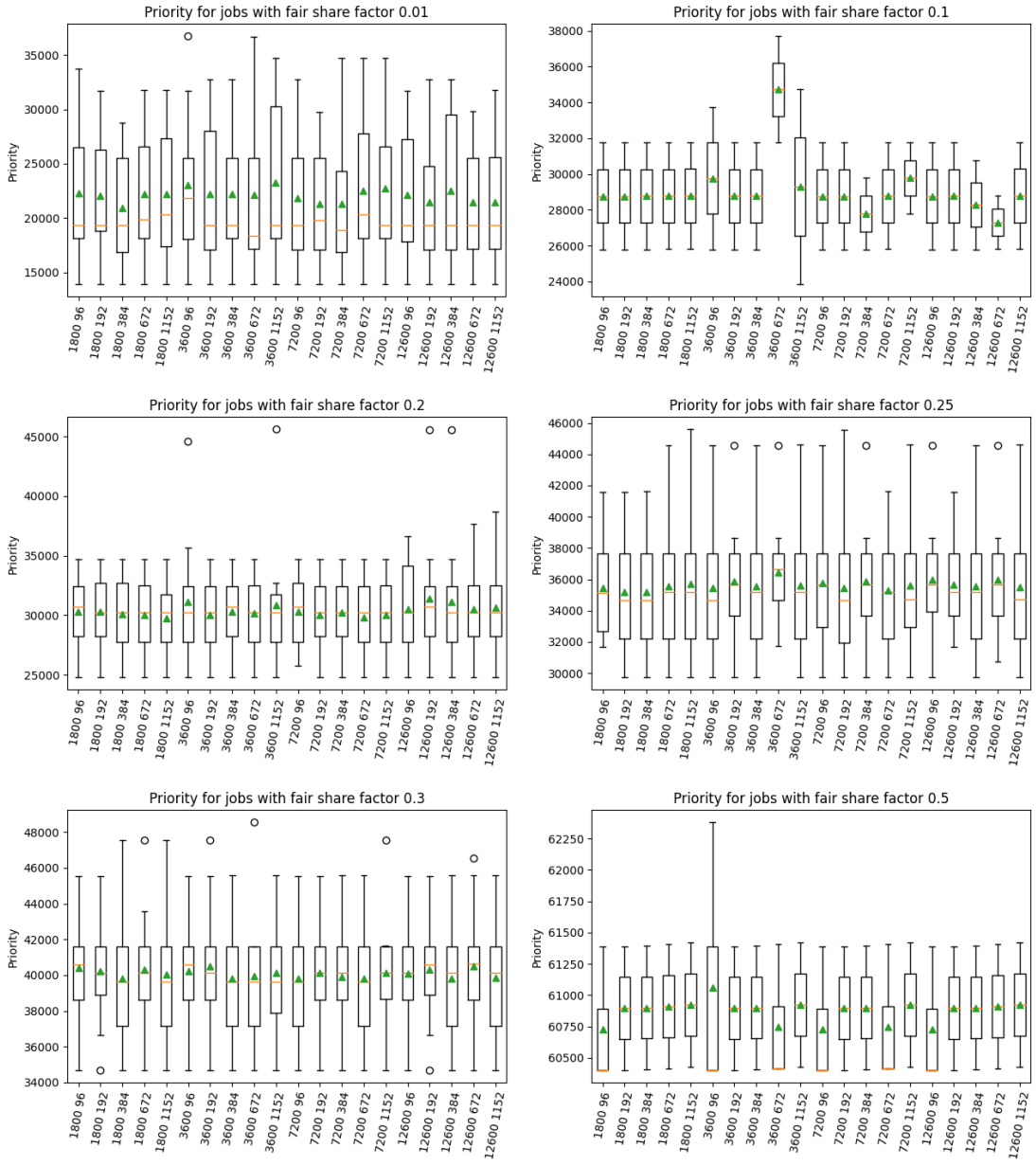

Figure 4.4: Box plot of the priority per fair share factor for values 0.01, 0.1, 0.2, 0.25, 0.3, and 0.5. Each box plot is a combination of CPU and runtime. Green triangle indicates the arithmetic average.

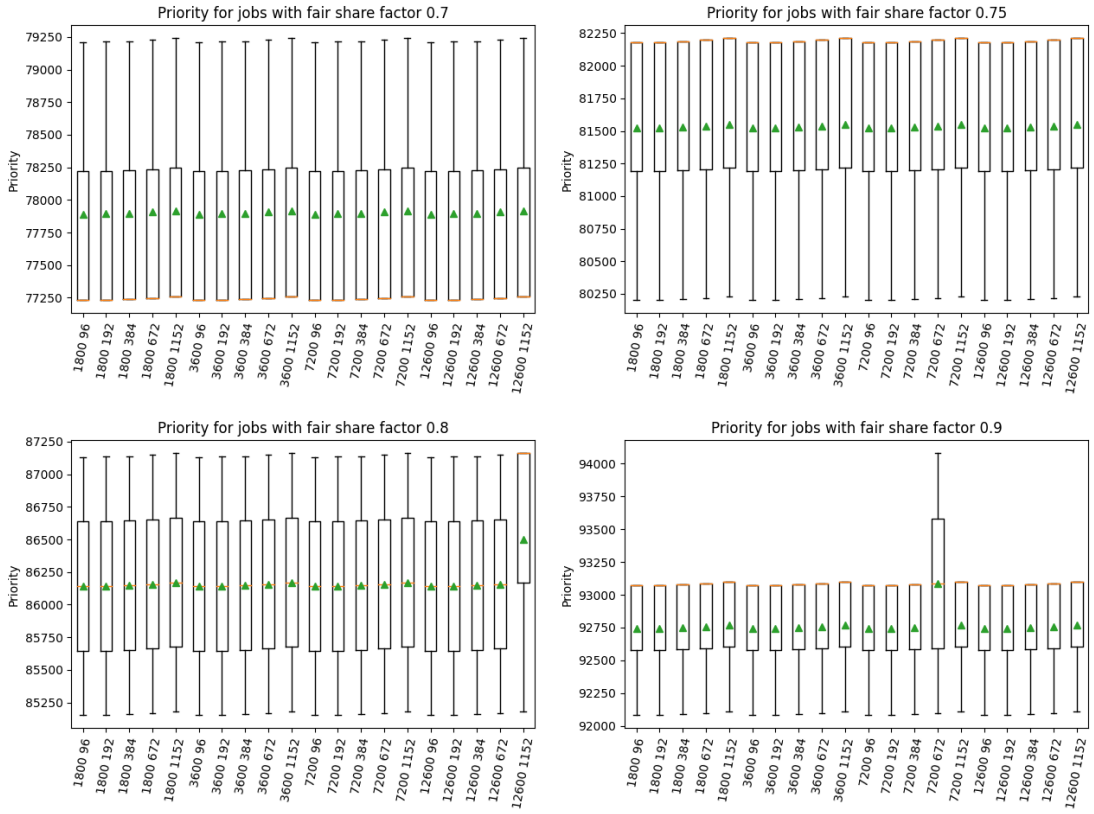

Figure 4.5: Box plot of the priority per fair share for factors 0.7, 0.75, 0.8, and 0.9. Each box plot is a combination of CPU and runtime. Green triangle indicates the arithmetic average.

4.2 Dynamic Workload

As explained in subsection 3.3.2, in this section we are tracking a full workflow example for each wrapper type: vertical and horizontal.

In tables 4.2 and 4.3 we show the total waiting time of the workflow per fair share value and submission time. We also compute the relative improvement of the aggregation.

We see that, for the best fair share factor, wrapping *never increases the waiting time* in both workflows: vertical and horizontal. In relative terms, we observe improvement of up to 100% in the queue time, in the vertical workflow submitted in the in instant A.2 in table 4.2, according to the labelling from table 3.24. But the best performing in absolute terms was the horizontal wrapper submitted at the instant B.2 in table 4.3, with a reduction of 1520 seconds in the queue time.

Under the worst fair share factor configuration, we observe that the vertical wrappers in table 4.2 reduce substantially the waiting time, from relative improvements of up to 98 % in all but two submission moments: at instants A.1 and B.1, where we observe an increase of at most 1.5 times. As for the horizontal workflow, we see that the wrapping improved with various degrees relatively: 7 %, 21 %, and 97 % for the submissions at instants A.1, A.2, B.1, and B.2. While both A.3 and B.3 instants show an increase in the wait time of up to 10304 seconds in absolute values.

Fair share	Label	Unwrapped T_q (s)	Wrapped T_q (s)	Δ (s)	Improvement (%)
1.0	A.1	0	0	0	0.00
1.0	A.2	134	4	130	97.01
1.0	A.3	0	0	0	0.00
1.0	B.1	0	0	0	0.00
1.0	B.2	221	0	221	100.00
1.0	B.3	0	0	0	0.00
6.9e-03	A.1	1047	1294	-247	-23.59
6.9e-03	A.2	249	8	241	96.79
6.9e-03	A.3	183	3	180	98.36
6.9e-03	B.1	291	416	-125	-42.95
6.9e-03	B.2	1323	167	1156	87.38
6.9e-03	B.3	5710	1504	4206	73.66

Table 4.2: Vertical workflow queue time results. T_q is the total time in queue of workflow.

Fair share	Label	Unwrapped T_q (s)	Wrapped T_q (s)	Δ (s)	Improvement (%)
1.0	A.1	0	0	0	0.00
1.0	A.2	126	65	61	48.41
1.0	A.3	0	0	0	0.00
1.0	B.1	0	0	0	0.00
1.0	B.2	1927	407	1520	78.88
1.0	B.3	0	0	0	0.00
6.9e-03	A.1	2453	2270	183	7.46
6.9e-03	A.2	557	435	122	21.90
6.9e-03	A.3	3	857	-854	-28466.67
6.9e-03	B.1	50	50	0	0.00
6.9e-03	B.2	9800	289	9511	97.05
6.9e-03	B.3	1265	11569	-10304	-814.54

Table 4.3: Horizontal workflow queue time results for dynamic workloads. T_q is the total time in queue of workflow.

Chapter 5

Discussion

Having new data from consolidated systems was a prerequisite for proving that our technique could decrease the time in queue of the execution of the overall workflow. Unfortunately, at Feitelson's repository [21] the machines available were either too old or too small, i.e. few CPUs and/or used only by a handful of users, to be comparable with machines in the same context as MareNostrum 4. What we saw is that most machines in Feitelson's repository were not constantly occupied as we see in the platforms we are interested in, like the usage observed in figure 1.1.

In such conditions we had only three options: alter the workload, reduce the resources of the simulated machine, or to increase the required resources of the workflow in order to *introduce artificially* a distressed enough scenario. All three of them could be *equally criticised for granting the means for the thesis we want to prove*. The importance of workloads and their design is commented thoroughly in [18], where the author talks about how not resembling reality leads to irrelevant results. So, in this work, we particularly rigorous with dealing with the data. We settled with a minimally cleaned by Feitelson workload, from the CEA-Curie machine; fit distributions to the data as seen in the LUMI supercomputer without any cleansing whatsoever of the trace; and, from the workflow side, we used a typical workflow as run in the Earth Science department of the BSC.

We tried to get newer data. We asked authors of [16] for the raw data of the workload of their machine, Nurion, only to be declined under "security concerns". Moreover, our institution's supercomputer, MareNostrum 4, does not have its data publicly available. Hence, we had to settle with an old system, CEA-Curie and its low number of congested moments.

LUMI, in the other hand, has only just entered into production. The machine was available for users in the beginning of 2023. And, as one might expect from any flagship in the bleeding edge of the technology, it is still receiving plenty of updates which might cause instabilities in the operation and, hence, some anomalies in the workload. Consequently, we *ruled out the usage of it for the dynamic case*, since the burstiness of the job submissions may very well be larger than on a matured system like MareNostrum 4. There are plenty of users testing the system and porting their work. And, as for the static workload, we used it with extreme caution since applications might be unstable in the newer hardware.

As for our results, we believe that those submissions where the overall waiting time increased with respect to the unaggregated case in tables 4.2 and 4.3 is due to the interaction with the backfill scheduler. The longer – more runtime – or the wider – more allocated CPUs – the job is, the harder it is to find a free spot for it that will not bother the execution of the other ones [7]. As a result, there is a high variability

depending on the current machine status. It could be that the machine just happened to have a new spot available that could be opportunistically used by the submitted job, which the aggregated case was too big to fit. This could explain the up to 7 hours more of wait time of wrapped jobs compared to their unwrapped counterpart, as seen in the submission on the B.3 submission instant for the horizontal workflow in table 4.3.

Also, our experiments were held under the simplified scenario of only a *single QoS* and *single partition*, where we have that backfill is responsible for more than 80% of the scheduling. In the other hand, backfilling in static workloads has to be analyzed carefully. Since they have one instant where the jobs are submitted, the simulator's window of utilization is short. That is, the machine will run out of jobs to execute and, consequently, empty itself. This impacts backfilling since if a job sits long enough in the queue, as this static workload simulation progresses, it will be ever so more likely to find a free spot. And that is in spite of the size in either dimension: CPU or runtime.

Moreover, when comparing the static and dynamic workloads, we have to bear in mind that both results *provide hints* to what are the jobs characteristics that have more impact. Static workload put the system on a short distressed environment, whereas the dynamic follow a real usage with cyclic stressed and calm periods. In the latter case, breaking the wrapper into the atomic jobs increases the odds of being backfilled. Even if the *workflow executing user* has a low fair share of $O(10^{-3})$.

Nevertheless, it was surprising to see the static workload's independence of the waiting time with respect to the allocated CPUs and runtime in table 4.1. We believe this could be due to the static workloads myopic property with respect to the evolution of the system, since there is no enough sustained usage on the machine.

Chapter 6

Conclusions

We conclude that aggregation, under the best fair share, *does reduce the overall execution time of the workflow*. As we see in the dynamic workloads results in tables 4.2 and 4.3. This is also supported by the static workload experiments, where we see an exponential reduction of the wait time with respect to the fair share from 0.1 onwards, clearly seen in figure 4.1. Under the worst fair share, aggregation benefit exceeds the eventual underperformance on the vertical workflow case, seen in table 4.2. Where we don't conclude a better performance is for the horizontal workflow case.

Then, with these conclusions, and bearing in mind the objective of *reducing the total time of execution of the workflow*, accounting for both runtime and queue time, we can draw strategies to be implemented in Autosubmit – seen in section 2.7 –. If the user the highest fair share, *it should submit tasks aggregated* and, if the user has a vertical wrapper, we also *recommend to use task aggregation* – regardless of the fair share.

Some of our results do not follow our thesis, namely submissions at instant A.3 and B.3 of the horizontal workflow under the very worst fair share seen in table 4.3. We expected the task aggregation to excel under such conditions, since it would save multiple submissions with very low priority. However, the interplay of the backfill and the quick increase in the fair share of the workflow executing user could be playing a bigger role on the waiting time and would need additional attention that could be subject of another study.

6.1 Future Work

In this area, experimenting is challenging. We have many different scenarios and plenty of "what if"s which grow considerably as we try to image situations under which the submission of a workflow might fall on. In the future, we can reduce the number of experimentation by building an analytical model, one using queue theory [59] which would allow us to have theoretical results and funnel the number of experiments with the simulator.

For the simulation, we have simplified the submission to just one QoS and one partition. System administrators are aware that Slurm benefits larger jobs, so it is common, and recommended by Slurm developers, to configure more QoSs restricted only to the small jobs. For reference, the largest weight factor for the priority in MareNostrum 4 is given to the QoS, $1e6$. This was not taken into account in our analysis, small jobs were submitted along side with the larger counterparts to the same QoS.

Different partitions are also growing in popularity among HPC centers. The era of a single homogeneous partition with CPUs is over. The current flagship machines

have different partitions for each technologies which users can take advantage of. As an example, LUMI has a partition of GPUs that is composed of more nodes than the one of pure CPUs. MareNostrum 5 [60] will have a similar setup, and also include other accelerators. These added features will have an impact on the scheduler, since it will have to account if the jobs require such features.

Aiming for better representations of the static workload, we could use a different approach to model it. Instead of doing a job-centric, that is, first generating the job allocated CPUs and runtime to then assign it to a user, we could invert this. Users in HPC centers tend to have a discrete set of configurations that they use [26]. Hence, we could first generate users and their usage pattern, and then draw for each of them the jobs for the workload.

But still in the job-centric approach, in this work we just fitted a probability distribution onto the allocated CPUs, without taking into account there are clusterings on jobs that are power of 2 [26] [18]. In our data, when we compare the log-log plots from the original job data, in figure 3.5, with the synthetic one, in figure 6.1, we notice how the methodology is short of capturing this structure, that appears as clear vertical lines with very few dots in between them. Besides improving the allocated CPUs, we could extend our analysis and create dynamic synthetic workloads following the methodology exposed in [19] to account for the inter-arrival time of jobs.

Additionally, Patel et al. [26] reported many trends on Mira and Intrepid. In the future, we could collect data from LUMI again, certainly maturer and with less variability to see if the user trends that they have reported still hold in the newer systems.

During the development of this work, we developed the *builtin* scheduler to be used by the simulator. This would allow us to disable the backfill by having the second thread 2.2 try to schedule jobs with respect to the order in priority only, as seen in subsection 2.2.1. Unfortunately, our implementation behaved in unpredictable ways. Hence we could validate our implementation in the same model as the BSC Slurm Simulator [23].

As for the workflow side, we could increase the number of tasks for longer workflows. It is not uncommon to scientists in the Earth Science department of the BSC to run workflows with more than 1000 tasks.

Finally, no analysis can be made without proper data. To address HPC center's reluctance to provide it, we could build a tool within Slurm that anonymizes in a rigorous way called Differential Privacy [61]. These techniques are used to make private data public in such a way that the aggregated is still useful, but without exposing a user's data. This concept is utilized by big technology companies [62] [63] [64] to process user data.

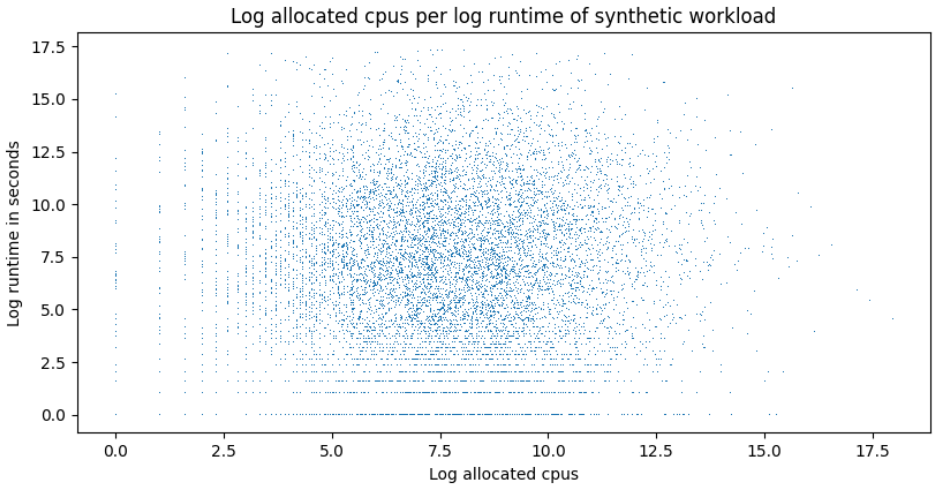

Figure 6.1: Scatter plot of the log allocated CPUs and log runtime of the static workloads used here.

References

[1] Martina Klose et al. "Mineral dust cycle in the Multiscale Online Nonhydrostatic AtmospheRe CHemistry model (MONARCH) version 2.0". In: *Geoscientific Model Development* 14.10 (2021), pp. 6403–6444.

[2] Benjamin P Abbott et al. "Full band all-sky search for periodic gravitational waves in the O1 LIGO data". In: *Physical Review D* 97.10 (2018), p. 102003.

[3] Mitchell R Vollger et al. "Increased mutation and gene conversion within human segmental duplications". In: *Nature* 617.7960 (2023), pp. 325–334.

[4] *MareNostrum*. URL: https://www.bsc.es/marenostrum/marenostrum/mn1 (visited on 09/30/2023).

[5] *The IBM100 - Blue Gene*. URL: https://www.ibm.com/ibm/history/ibm100/us/en/icons/bluegene/ (visited on 09/30/2023).

[6] *The K computer*. URL: https://www.riken.jp/en/collab/resources/kcomputer/ (visited on 09/30/2023).

[7] Srividya Srinivasan et al. "Selective reservation strategies for backfill job scheduling". In: *Job Scheduling Strategies for Parallel Processing: 8th International Workshop, JSSPP 2002 Edinburgh, Scotland, UK, July 24, 2002 Revised Papers 8*. Springer. 2002, pp. 55–71.

[8] David Glesser, Yiannis Georgiou, and Denis Trystram. "Introducing Energy based fair-share scheduling". In: *Slurm User Group Meeting 2014*. Lugano, Switzerland, 2014. URL: https://hal.science/hal-01102291.

[9] Emmanuel Jeannot, Guillaume Pallez, and Nicolas Vidal. "IO-aware Job-Scheduling: Exploiting the Impacts of Workload Characterizations to select the Mapping Strategy". In: *The International Journal of High Performance Computing Applications* (2023).

[10] Andy B Yoo, Morris A Jette, and Mark Grondona. "Slurm: Simple linux utility for resource management". In: *Job Scheduling Strategies for Parallel Processing: 9th International Workshop, JSSPP 2003, Seattle, WA, USA, June 24, 2003. Revised Paper 9*. Springer. 2003, pp. 44–60.

[11] URL: https://www.bsc.es/marenostrum/marenostrum (visited on 09/29/2023).

[12] URL: https://docs.lumi-supercomputer.eu/ (visited on 09/29/2023).

[13] URL: https://www.dkrz.de/en/systems/hpc (visited on 09/29/2023).

[14] URL: https://luxembourg.public.lu/en/invest/innovation/meluxina-supercomputer.html (visited on 09/29/2023).

[15] Judy Kay and Piers Lauder. "A fair share scheduler". In: *Communications of the ACM* 31.1 (1988), pp. 44–55.

[16] Ju-Won Park, Min-Woo Kwon, and Taeyoung Hong. "Queue congestion prediction for large-scale high performance computing systems using a hidden Markov model". In: *The Journal of Supercomputing* 78.10 (2022), pp. 12202–12223.

[17] Allen B Downey. "Using queue time predictions for processor allocation". In: *Job Scheduling Strategies for Parallel Processing: IPPS'97 Processing Workshop Geneva, Switzerland, April 5, 1997 Proceedings 3*. Springer. 1997, pp. 35–57.

[18] DG Feitelson. "Workload characterization and modeling book. Version 0.36". In: *School of Computer Science and Engineering, The Hebrew University of Jerusalem* (2012).

[19] Walfredo Cirne and Francine Berman. "Adaptive selection of partition size for supercomputer requests". In: *Job Scheduling Strategies for Parallel Processing: IPDPS 2000 Workshop, JSSPP 2000 Cancun, Mexico, May 1, 2000 Proceedings 6*. Springer. 2000, pp. 187–207.

[20] Steve J Chapin et al. "Benchmarks and standards for the evaluation of parallel job schedulers". In: *Job Scheduling Strategies for Parallel Processing: IPPS/SPDP'99Workshop, JSSPP'99 San Juan, Puerto Rico, April 16, 1999 Proceedings 5*. Springer. 1999, pp. 67–90.

[21] D. G. Feitelson and D. Tsafrir. *Logs of real parallel workloads from production systems*. Sept. 2019. URL: https://www.cs.huji.ac.il/labs/parallel/workload/logs.html (visited on 09/29/2023).

[22] Mario Acosta, Sergi Palomas, and Stella Paronuzzi. *IS-ENES3 D4.3 - CPMIP performance metrics and community advice*. Dec. 2021. DOI: 10.5281/zenodo.6394049. URL: https://doi.org/10.5281/zenodo.6394049.

[23] Marco D'Amico Ana Jokanovic and Julita Corbalan. "Evaluating SLURM Simulator with Real-Machine SLURM and Vice Versa". In: *Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS18) At: ACM/IEEE Supercomputing 2018* (2018).

[24] Nikolay A Simakov et al. "A slurm simulator: Implementation and parametric analysis". In: *High Performance Computing Systems. Performance Modeling, Benchmarking, and Simulation: 8th International Workshop, PMBS 2017, Denver, CO, USA, November 13, 2017, Proceedings 8*. Springer. 2018, pp. 197–217.

[25] Dirk Merkel. "Docker: lightweight linux containers for consistent development and deployment". In: *Linux journal* 2014.239 (2014), p. 2.

[26] Tirthak Patel et al. "Job characteristics on large-scale systems: long-term analysis, quantification, and implications". In: *SC20: International conference for high performance computing, networking, storage and analysis*. IEEE. 2020, pp. 1–17.

[27] Nick Brown et al. "Predicting batch queue job wait times for informed scheduling of urgent HPC workloads". In: *arXiv preprint arXiv:2204.13543* (2022).

[28] J. Kok Konjaang and Lina Xu. "Multi-objective workflow optimization strategy (MOWOS) for cloud computing". In: *Journal of Cloud Computing* 10.1 (Jan. 2021), p. 11. ISSN: 2192-113X. DOI: 10.1186/s13677-020-00219-1. URL: https://doi.org/10.1186/s13677-020-00219-1.

[29] SchedMD. *Network Configuration Guide*. Blog post. Nov. 2020. URL: https://slurm.schedmd.com/network.html (visited on 09/29/2023).

[30] *Scheduling Configuration Guide*. Jan. 2022. URL: https://slurm.schedmd.com/sched_config.html (visited on 09/29/2023).

[31] Danny Dolev et al. "No Justified Complaints: On Fair Sharing of Multiple Resources". In: *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*. ITCS '12. Cambridge, Massachusetts: Association for Computing Machinery, 2012, pp. 68–75. ISBN: 9781450311151. DOI: 10.1145/2090236.2090243. URL: https://doi.org/10.1145/2090236.2090243.

[32] Art Sedighi, Yuefan Deng, and Peng Zhang. "Fairness of task scheduling in high performance computing environments". In: *Scalable Computing: Practice and Experience* 15.3 (2014), pp. 271–288.

[33] SchedMD. *Fair Tree Fairshare Algorithm*. Blog post. Jan. 2019. URL: https://slurm.schedmd.com/fair_tree.html (visited on 09/29/2023).

[34] Dan Tsafrir, Yoav Etsion, and Dror G Feitelson. "Modeling user runtime estimates". In: *Job Scheduling Strategies for Parallel Processing: 11th International Workshop, JSSPP 2005, Cambridge, MA, USA, June 19, 2005, Revised Selected Papers 11*. Springer. 2005, pp. 1–35.

[35] Karim Baïna and Salah Baïna. "User experience-based evaluation of open source workflow systems: The cases of Bonita, Activiti, jBPM, and Intalio". In: *2013 3rd International Symposium ISKO-Maghreb*. 2013, pp. 1–8. DOI: 10.1109/ISKO-Maghreb.2013.6728122.

[36] R. Döscher et al. "The EC-Earth3 Earth system model for the Coupled Model Intercomparison Project 6". In: *Geoscientific Model Development* 15.7 (2022), pp. 2973–3020. DOI: 10.5194/gmd-15-2973-2022. URL: https://gmd.copernicus.org/articles/15/2973/2022/.

[37] Günther Zängl et al. "The ICON (ICOsahedral Non-hydrostatic) modelling framework of DWD and MPI-M: Description of the non-hydrostatic dynamical core". In: *Quarterly Journal of the Royal Meteorological Society* 141.687 (2015), pp. 563–579.

[38] Kristian Mogensen, Sarah Keeley, and Peter Towers. *Coupling of the NEMO and IFS models in a single executable*. Vol. 10. ECMWF Reading, United Kingdom, 2012.

[39] Domingo Manubens-Gil et al. "Seamless management of ensemble climate prediction experiments on HPC platforms". In: *2016 International Conference on High Performance Computing and Simulation (HPCS)*. 2016, pp. 895–900. DOI: 10.1109/HPCSim.2016.7568429.

[40] Hilary J. Oliver, Matthew Shin, and Oliver Sanders. "Cylc: A Workflow Engine for Cycling Systems". In: *Journal of Open Source Software* 3.27 (2018), p. 737. DOI: 10.21105/joss.00737. URL: https://doi.org/10.21105/joss.00737.

[41] Ewa Deelman et al. "The Evolution of the Pegasus Workflow Management Software". In: *Computing in Science & Engineering* 21.4 (2019), pp. 22–36. DOI: 10.1109/MCSE.2019.2919690.

[42] Francesc Lordan et al. "Servicess: An interoperable programming framework for the cloud". In: *Journal of grid computing* 12 (2014), pp. 67–91.

[43] Felix Mölder et al. "Sustainable data analysis with Snakemake". In: *F1000Research* 10 (2021).

[44] C. Pérez et al. "Atmospheric dust modeling from meso to global scales with the online NMMB/BSC-Dust model". In: *Atmospheric Chemistry and Physics* 11.24 (2011), pp. 13001–13027. DOI: 10.5194/acp-11-13001-2011. URL: https://acp.copernicus.org/articles/11/13001/2011/.

[45] URL: https://www.ibm.com/docs/en/spectrum-lsf/10.1.0?topic=lsf-session-scheduler (visited on 09/29/2023).

[46] URL: https://altair.com/pbs-professional (visited on 09/29/2023).

[47] URL: https://gridscheduler.sourceforge.net/ (visited on 09/29/2023).

[48] URL: https://software.fujitsu.com/jp/manual/manualfiles/m220008/j2ul2452/02enz007/j2ul-2452-02enz0.pdf (visited on 09/29/2023).

[49] Andrea Manrique-Suñén et al. "Subseasonal predictions for climate services, a recipe for operational implementation". In: *Climate Services* 30 (2023), p. 100359. ISSN: 2405-8807. DOI: https://doi.org/10.1016/j.cliser.2023.100359. URL: https://www.sciencedirect.com/science/article/pii/S2405880723000201.

[50] *Meteorological Archival and Retrieval System*. URL: https://www.ecmwf.int/en/forecasts/access-forecasts/access-archive-datasets (visited on 10/04/2023).

[51] Raul Corredor Asenjo. *Cirrus: el nuevo sistema de supercomputación de AEMET*. URL: https://aemetblog.es/2021/04/27/cirrus-el-nuevo-sistema-de-supercomputacion-de-aemet/ (visited on 09/29/2023).

[52] *Destination Earth*. URL: https://destination-earth.eu/ (visited on 09/30/2023).

[53] *Digital Twin Oceans*. URL: https://digitaltwinocean.mercator-ocean.eu/ (visited on 09/30/2023).

[54] *European Eddy RIch Earth System Models*. URL: https://eerie-project.eu/ (visited on 09/30/2023).

[55] The pandas development team. *pandas-dev/pandas: Pandas*. Version latest. Feb. 2020. DOI: 10.5281/zenodo.3509134. URL: https://doi.org/10.5281/zenodo.3509134.

[56] Wes McKinney. "Data Structures for Statistical Computing in Python". In: *Proceedings of the 9th Python in Science Conference*. Ed. by Stéfan van der Walt and Jarrod Millman. 2010, pp. 56–61. DOI: 10.25080/Majora-92bf1922-00a.

[57] Pauli Virtanen et al. "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python". In: *Nature Methods* 17 (2020), pp. 261–272. DOI: 10.1038/s41592-019-0686-2.

[58] P. Stoica and Y. Selen. "Model-order selection: a review of information criterion rules". In: *IEEE Signal Processing Magazine* 21.4 (2004), pp. 36–47. DOI: 10.1109/MSP.2004.1311138.

[59] Mor Harchol-Balter. *Performance modeling and design of computer systems: queueing theory in action*. Cambridge University Press, 2013.

[60] Jordi Pérez Colomé. *Llega El Supercomputador marenostrum 5, la mayor Inversión Europea en una infraestructura científica en España*. Sept. 2023. URL: https://elpais.com/tecnologia/2023-09-12/llega-el-supercomputador-marenostrum-5-la-mayor-inversion-europea-en-una-infraestructura-cientifica-en-espana.html.

[61] Cynthia Dwork. "Differential Privacy: A Survey of Results". In: *Theory and Applications of Models of Computation*. Ed. by Manindra Agrawal et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 1–19. ISBN: 978-3-540-79228-4.

[62] Sarah Bird. *Putting differential privacy into practice to use data responsibly*. Blog post. URL: https://blogs.microsoft.com/ai-for-business/differential-privacy/ (visited on 09/29/2023).

[63] Apple Inc. *Differential Privacy*. Tech. rep. Apple Inc., Feb. 2017. URL: https://www.apple.com/privacy/docs/Differential_Privacy_Overview.pdf (visited on 09/29/2023).

[64] Meta Platforms Inc. *The value of differential privacy*. Blog post. URL: https://privacytech.fb.com/differential-privacy/ (visited on 09/29/2023).

Appendix A

Full Static Workloads Tables

CPUs	Fair share	$\overline{T_q}$ (s)	$max\ T_q$ (s)	$min\ T_q$ (s)	$\overline{P}$	$max\ P$	$min\ P$
96	0.01	304.3	507	131	22315.0	33725	13886
96	0.1	310.4	568	131	28769.5	31751	25788
96	0.2	157.9	306	70	30318.3	34678	24771
96	0.25	103.0	141	60	35460.3	41601	31700
96	0.3	66.4	123	9	40406.6	45555	34665
96	0.5	0.0	0	0	60729.0	61389	60399
96	0.7	0.0	0	0	77891.0	79211	77231
96	0.75	0.0	0	0	81521.0	82181	80201
96	0.8	0.0	0	0	86142.0	87132	85152
96	0.9	0.0	0	0	92742.0	93072	92082
192	0.01	304.3	507	131	22020.1	31746	13888
192	0.1	292.1	507	131	28767.5	31747	25788
192	0.2	157.9	306	70	30319.9	34679	24773
192	0.25	109.1	184	60	35165.8	41603	29723
192	0.3	66.4	123	9	40210.5	45557	34667
192	0.5	0.0	0	0	60896.0	61391	60401
192	0.7	0.0	0	0	77893.0	79213	77233
192	0.75	0.0	0	0	81523.0	82183	80203
192	0.8	0.0	0	0	86143.3	87134	85153
192	0.9	0.0	0	0	92744.0	93074	92084
384	0.01	298.2	568	131	20935.2	28787	13893
384	0.1	304.3	507	131	28773.5	31752	25795
384	0.2	145.7	202	70	30125.8	34685	24778
384	0.25	109.1	184	60	35170.9	41608	29728
384	0.3	66.4	123	9	39819.5	47542	34672
384	0.5	0.0	0	0	60901.0	61396	60406
384	0.7	0.0	0	0	77898.0	79218	77238
384	0.75	0.0	0	0	81528.0	82188	80208
384	0.8	0.0	0	0	86149.0	87139	85159
384	0.9	0.0	0	0	92749.0	93079	92089
672	0.01	304.3	507	131	22232.1	31760	13902
672	0.1	292.1	507	131	28782.5	31761	25804
672	0.2	145.7	202	70	30035.7	34694	24787
672	0.25	109.1	184	60	35575.9	44587	29737
672	0.3	66.4	123	9	40323.5	47551	34681

672	0.5	0.0	0	0	60910.0	61405	60415
672	0.7	0.0	0	0	77907.0	79227	77247
672	0.75	0.0	0	0	81537.0	82197	80217
672	0.8	0.0	0	0	86158.0	87148	85168
672	0.9	0.0	0	0	92758.0	93088	92098
1152	0.01	304.3	507	131	22244.6	31773	13915
1152	0.1	292.1	507	131	28794.5	31773	25816
1152	0.2	145.7	202	70	29751.3	34706	24799
1152	0.25	109.1	184	60	35687.3	45590	29749
1152	0.3	66.4	123	9	40038.8	47563	34693
1152	0.5	0.0	0	0	60922.5	61418	60427
1152	0.7	0.0	0	0	77919.0	79239	77259
1152	0.75	0.0	0	0	81549.7	82210	80229
1152	0.8	0.0	0	0	86170.0	87160	85180
1152	0.9	0.0	0	0	92771.0	93101	92111

Table A.1: LUMI static experiment results with 1800 seconds of runtime. T_q is the time in queue and P is the priority upon start time. The overline on top of the symbol indicates average over the 10 different workloads generated.

CPUs	Fair share	$\overline{T_q}$ (s)	$max\ T_q$ (s)	$min\ T_q$ (s)	$\overline{P}$	$max\ P$	$min\ P$
96	0.01	328.7	794	131	23010.7	36726	13886
96	0.1	286.0	568	131	29760.0	33732	25788
96	0.2	151.8	245	70	31109.6	44586	24771
96	0.25	109.1	184	60	35460.9	44571	29721
96	0.3	66.4	123	9	40208.5	45555	34665
96	0.5	0.0	0	0	61059.0	62379	60399
96	0.7	0.0	0	0	77891.0	79211	77231
96	0.75	0.0	0	0	81521.0	82181	80201
96	0.8	0.0	0	0	86142.0	87132	85152
96	0.9	0.0	0	0	92742.0	93072	92082
192	0.01	316.5	672	131	22219.2	32754	13888
192	0.1	298.2	568	131	28771.0	31753	25789
192	0.2	145.7	202	70	30021.6	34679	24773
192	0.25	109.1	184	60	35859.1	44574	29723
192	0.3	66.4	123	9	40507.4	47537	34666
192	0.5	0.0	0	0	60896.0	61391	60401
192	0.7	0.0	0	0	77893.0	79213	77233
192	0.75	0.0	0	0	81523.0	82183	80203
192	0.8	0.0	0	0	86143.3	87134	85153
192	0.9	0.0	0	0	92744.0	93074	92084
384	0.01	310.4	507	131	22223.8	32739	13893
384	0.1	304.3	568	131	28776.5	31758	25795
384	0.2	157.9	306	70	30325.4	34685	24778
384	0.25	109.1	184	60	35567.0	44578	29728
384	0.3	66.4	123	9	39819.5	45562	34672
384	0.5	0.0	0	0	60901.0	61396	60406
384	0.7	0.0	0	0	77898.0	79218	77238
384	0.75	0.0	0	0	81528.0	82188	80208

384	0.8	0.0	0	0	86149.0	87139	85159
384	0.9	0.0	0	0	92749.0	93079	92089
672	0.01	310.4	611	131	22133.9	36723	13902
672	0.1	328.7	794	131	34726.5	37692	31761
672	0.2	145.7	202	70	30134.8	34694	24787
672	0.25	115.2	245	60	36467.7	44587	31716
672	0.3	66.4	123	9	39927.5	48541	34681
672	0.5	0.0	0	0	60745.0	61405	60415
672	0.7	0.0	0	0	77907.0	79227	77247
672	0.75	0.0	0	0	81537.0	82197	80217
672	0.8	0.0	0	0	86158.0	87148	85168
672	0.9	0.0	0	0	92758.0	93088	92098
1152	0.01	322.6	568	131	23236.8	34746	13915
1152	0.1	310.4	568	131	29293.5	34751	23836
1152	0.2	151.8	245	70	30840.9	45604	24799
1152	0.25	109.1	184	60	35588.5	44600	29750
1152	0.3	60.3	80	9	40137.1	45583	34693
1152	0.5	0.0	0	0	60922.5	61418	60427
1152	0.7	0.0	0	0	77919.0	79239	77259
1152	0.75	0.0	0	0	81549.7	82210	80229
1152	0.8	0.0	0	0	86170.0	87160	85180
1152	0.9	0.0	0	0	92771.0	93101	92111

Table A.2: LUMI static experiment results with 3600 seconds of runtime. T_q is the time in queue and P is the priority upon start time. The overline on top of the symbol indicates average over the 10 different workloads generated.

CPUs	Fair share	$\overline{T_q}$ (s)	$max\ T_q$ (s)	$min\ T_q$ (s)	$\overline{P}$	$max\ P$	$min\ P$
96	0.01	304.3	568	131	21820.0	32732	13886
96	0.1	334.8	855	131	28766.0	31744	25788
96	0.2	145.7	202	70	30316.7	34678	25760
96	0.25	103.0	141	60	35757.4	44571	29721
96	0.3	66.4	123	9	39812.5	45554	34665
96	0.5	0.0	0	0	60729.0	61389	60399
96	0.7	0.0	0	0	77891.0	79211	77231
96	0.75	0.0	0	0	81521.0	82181	80201
96	0.8	0.0	0	0	86142.0	87132	85152
96	0.9	0.0	0	0	92742.0	93072	92082
192	0.01	292.1	568	131	21325.5	29773	13888
192	0.1	292.1	507	131	28768.5	31747	25790
192	0.2	157.9	306	70	30022.9	34679	24773
192	0.25	103.0	141	60	35462.2	45563	29723
192	0.3	66.4	123	9	40111.3	45556	34666
192	0.5	0.0	0	0	60896.0	61391	60401
192	0.7	0.0	0	0	77893.0	79213	77233
192	0.75	0.0	0	0	81523.0	82183	80203
192	0.8	0.0	0	0	86143.3	87134	85153
192	0.9	0.0	0	0	92744.0	93074	92084
384	0.01	310.4	672	131	21332.7	34741	13893

384	0.1	322.6	672	131	27787.0	29779	25795
384	0.2	157.9	306	70	30226.3	34685	24778
384	0.25	109.1	184	60	35864.2	44579	29728
384	0.3	66.4	123	9	39918.5	45562	34672
384	0.5	0.0	0	0	60901.0	61396	60406
384	0.7	0.0	0	0	77898.0	79218	77238
384	0.75	0.0	0	0	81528.0	82188	80208
384	0.8	0.0	0	0	86149.0	87139	85159
384	0.9	0.0	0	0	92749.0	93079	92089
672	0.01	316.5	507	131	22530.4	34728	13902
672	0.1	316.5	672	131	28782.5	31761	25804
672	0.2	145.7	202	70	29837.8	34694	24787
672	0.25	115.2	245	60	35279.6	41617	29737
672	0.3	66.4	123	9	39828.4	45571	34680
672	0.5	0.0	0	0	60745.0	61405	60415
672	0.7	0.0	0	0	77907.0	79227	77247
672	0.75	0.0	0	0	81537.0	82197	80217
672	0.8	0.0	0	0	86158.0	87148	85168
672	0.9	0.0	0	0	93088.0	94078	92098
1152	0.01	316.5	550	131	22741.1	34746	13915
1152	0.1	310.4	507	131	29784.5	31773	27796
1152	0.2	145.7	202	70	30048.2	34706	24799
1152	0.25	103.0	184	60	35587.8	44600	29750
1152	0.3	66.4	123	9	40138.1	47563	34693
1152	0.5	0.0	0	0	60922.5	61418	60427
1152	0.7	0.0	0	0	77919.0	79239	77259
1152	0.75	0.0	0	0	81549.7	82210	80229
1152	0.8	0.0	0	0	86170.0	87160	85180
1152	0.9	0.0	0	0	92771.0	93101	92111

Table A.3: LUMI static experiment results with 7200 seconds of runtime. T_q is the time in queue and P is the priority upon start time. The overline on top of the symbol indicates average over the 10 different workloads generated.

CPUs	Fair share	$\overline{T_q}$ (s)	$max\ T_q$ (s)	$min\ T_q$ (s)	$\overline{P}$	$max\ P$	$min\ P$
96	0.01	292.1	507	131	22115.6	31744	13886
96	0.1	304.3	550	131	28766.0	31744	25788
96	0.2	145.7	202	70	30514.9	36650	24771
96	0.25	115.2	245	60	35956.7	44571	29721
96	0.3	66.4	123	9	40109.5	45555	34665
96	0.5	0.0	0	0	60729.0	61389	60399
96	0.7	0.0	0	0	77891.0	79211	77231
96	0.75	0.0	0	0	81521.0	82181	80201
96	0.8	0.0	0	0	86142.0	87132	85152
96	0.9	0.0	0	0	92742.0	93072	92082
192	0.01	316.5	611	131	21427.5	32749	13888
192	0.1	316.5	611	131	28772.0	31754	25790
192	0.2	151.8	245	70	31408.2	45578	24773
192	0.25	109.1	184	60	35660.8	41603	31702

192	0.3	66.4	123	9	40309.4	45557	34666
192	0.5	0.0	0	0	60896.0	61391	60401
192	0.7	0.0	0	0	77893.0	79213	77233
192	0.75	0.0	0	0	81523.0	82183	80203
192	0.8	0.0	0	0	86143.3	87134	85153
192	0.9	0.0	0	0	92744.0	93074	92084
384	0.01	304.3	550	131	22520.4	32747	13893
384	0.1	304.3	568	131	28282.0	30769	25795
384	0.2	151.8	245	70	31116.6	45583	24778
384	0.25	115.2	245	60	35567.7	44578	29728
384	0.3	66.4	123	9	39819.7	45562	34672
384	0.5	0.0	0	0	60901.0	61396	60406
384	0.7	0.0	0	0	77898.0	79218	77238
384	0.75	0.0	0	0	81528.0	82188	80208
384	0.8	0.0	0	0	86149.0	87139	85159
384	0.9	0.0	0	0	92749.0	93079	92089
672	0.01	292.1	568	131	21438.6	29787	13902
672	0.1	359.2	916	131	27300.5	28797	25804
672	0.2	157.9	306	70	30532.2	37664	24787
672	0.25	109.1	184	60	35972.2	44588	30727
672	0.3	66.4	123	9	40521.5	46561	34681
672	0.5	0.0	0	0	60910.0	61405	60415
672	0.7	0.0	0	0	77907.0	79227	77247
672	0.75	0.0	0	0	81537.0	82197	80217
672	0.8	0.0	0	0	86158.0	87148	85168
672	0.9	0.0	0	0	92758.0	93088	92098
1152	0.01	292.1	568	131	21451.2	31780	13915
1152	0.1	304.3	507	131	28794.0	31772	25816
1152	0.2	157.9	306	70	30643.6	38667	24799
1152	0.25	109.1	184	60	35489.5	44600	29750
1152	0.3	66.4	123	9	39840.8	45583	34693
1152	0.5	0.0	0	0	60922.5	61418	60427
1152	0.7	0.0	0	0	77919.0	79239	77259
1152	0.75	0.0	0	0	81549.7	82210	80229
1152	0.8	0.0	0	0	86500.0	87160	85180
1152	0.9	0.0	0	0	92771.0	93101	92111

Table A.4: LUMI static experiment results with 12600 seconds of runtime. T_q is the time in queue and P is the priority upon start time. The overline on top of the symbol indicates average over the 10 different workloads generated.