The High Performance Scheduler Game: A Characterization of Slurm, Metrics, and the Viability of Cooperation

Wilmer V. Uruchi Ticona

July 2020

Advisor: Maria Jose Serna Iglesias
Computer Science Department

MASTER IN INNOVATION AND RESEARCH IN INFORMATICS
Advanced Computing

FACULTAT D'INFORMÀTICA DE BARCELONA (FIB)
UNIVERSITAT POLITÈCNICA DE CATALUNYA (UPC) – BarcelonaTech

Abstract

The *Slurm* Scheduler is a widely used tool for scheduling in High Performance Computing platforms around the world. Several studies have been conducted to find ways to improve the response time of the system, among other specific performance metrics; mainly from an algorithmic perspective. Scheduling has also been studied from the viewpoint of *Game Theory*, where models that attempt to capture the main characteristics of the problem are developed and analyzed. In this study, we have used the tools that *Algorithmic Game Theory* provides, specifically from the field of *Mechanism Design*, to develop and study a model that captures some of the main characteristics of the *Slurm* Scheduler. We developed the necessary software to test these models, we document it, and provide it through a public online repository so anyone can reproduce our results. We performed a thorough data analysis process to build a reliable data source based on real usage information from a workflow manager used at *Barcelona Supercoming Center*, the **Autosubmit** library. The whole data mining process is documented in this study, and the necessary code has been included in the online repository. Then, through experimentation, we analyzed how our model and its variants behave; furthermore, we compared these results with the results from an existing *Slurm* Simulator, developed by *Barcelona Supercomputing Center* members. Using these results, we calculated an approximate value for the *Price of Anarchy*, and we discuss the *Viability of Cooperation* in the context of the *Slurm* Scheduler.

Contents

1 Introduction

In the context of a *High Performance Computing* cluster, scheduling is a necessary operation that ensures that there exists an acceptable quality of service for the variety of users of the limited resources of the cluster. The scheduling process can vary from a simple *First Comes First Served* model to a wide variety of more complex implementations that tend to satisfy specific requirements from each group of users.

Slurm is an open source, fault-tolerant, and highly scalable cluster management system for large and small Linux clusters [12]. It offers a wide array of plugins and configuration options that are used by the platform administrators to best distribute the resources at hand. *MareNostrum4*, a High Performance Computer located in Barcelona (next to the *Campus Nord* of the *Universitat Politècnica de Catalunya*) and managed by the *Barcelona Supercomputing Center - Centro Nacional de Supercomputación* (*BSC-CNS*), implements *Slurm* to manage the allocation of resources to the jobs send to it by a variety of users [7]. These users execute different types of computer simulation models or other computational tasks that usually implement high levels of parallelism.

The **main motivation** of this study is to get a better understanding of the *Slurm* Scheduler mechanism, since it is one of the most widely used management platform for High Performance Computing clusters around the world. The users of these clusters are usually involved in complex projects of key importance for different areas of human development. For example, the *Department of Earth Sciences*, among other concerns, is studying the effects of human activity in climate change around the world, present and future. Then, by studying the *Slurm* Scheduler, we hope to indirectly help those groups of scientists that make use of the HPC platforms managed by this scheduler.

The **main objective** of this study is to analyze whether *Algorithmic Game Theory* and specifically **Mechanism Design** tools can be used to build a model that represents some key characteristics of the *Slurm* Scheduler. We aim at producing a model that implements some desired guarantees, so we can achieve optimal results that can be compared with variations of the model and even with the results of a proper ***Slurm* Simulator** [1]. The inclusion of *Mechanism Design* is key to this objective because this field gives us the tools to design an ideal mechanism that can produce an optimal result, and also the tools to perform a correct analysis of it. Furthermore, the *Algorithmic Game Theory* approach gives us an economic perspective that we can use to understand the effect of user interaction and selfish behavior into a model where users compete for limited resources.

In order to get a better knowledge of the behavior of the scheduler and the causes of it, we perform experimentation using implementations of the model we propose and its variants. We also experiment using the *Slurm* Simulator and compare its results with our results using a defined common ground. By doing this, we also try to get a glimpse at the **Price of Anarchy** of the *Slurm* Scheduler, which is the ratio between the worst *possible* outcome and the best *possible* outcome. We present proper definitions in the corresponding sections.

Attempting to get an exact bound on the Price Anarchy for a mechanism as complex as the *Slurm* Scheduler would be an almost insurmountable task; however, we can play with our models to get an idea or at least some intuition of it. Then, adding upon the objectives previously described, we try to argue about the **Viability of Cooperation**.

As a **secondary objective**, we hope that the results and conclusions presented in this study can help the *Slurm* developers to get more ideas into which way to take the improvement of their platform. Moreover, we also hope that this study, across its different sections, can help users of the HPC platforms that implement *Slurm* to get a better idea of what is happening behind the scheduling process, so they can, perhaps, implement strategies of their own.

Another important **secondary objective** is to present an intuitive explanation of **Mechanism Design** concepts by testing them in an experimental setting, where we will be able to show the results predicted by our theoretic assumptions. We have seen that some of these concepts might be hard to grasp at the beginning, but are then better understood when presented along case studies. We organize and present these concepts in an intuitive way. Then, we present our experimentation as a case study where we showcase these concepts and analyze their interaction with the results obtained.

We also perform an in-depth analysis of the data generated by Autosubmit [3], a workflow manager used by the *Earth Science Department* at *BSC-CNS*. The **objective** of this task is two-fold: First, to get a detailed view of the usage of the library and its trend; second, to get a proper data source that could be used for experimentation.

The modeling task does **not only** include a theoretical formulation of a model that captures the main characteristics of the scheduler and subsequent discussion, but also the development of a specific testing platform appropriate for this model and our assumptions. It is important to clarify that this testing platform is not a *Slurm* Simulator, but a platform designed for our own purposes. We also make use of the *Slurm* Simulator for further experimentation and comparison.

We start by analyzing *Slurm* according to the documentation presented in their web page, the code implementation, and the experience of *BSC-CNS* users. We present an overview of the *Slurm* scheduler in Section 2 where we review the main components of the scheduling operation. In Section 2.1, we present a description of the user hierarchy present in the system. In Section 2.2, we present a description of the main factors involved in the scheduling operation, and describe some of the main algorithms used for the calculation of these factors, using examples where we consider further explanation is necessary or where we seem they were lacking in official documentation. Section 3 presents a somewhat formal description of the scheduling mechanism used in *Slurm* in an attempt to present it as clearly as possible using basic mathematical notation. Furthermore, we describe some variables that we consider play an important role in the scheduling mechanism.

In Section 4, we go through the data mining process required to produce a proper dataset for experimentation. Section 4.1 presents a brief overview of

our data source. Section 4.2 describes the main steps taken to generate the raw dataset. Section 4.4 is the bulk of the data analysis where we start by looking at the distribution of the experiments sent to the HPC and registered into the starting raw dataset. Then, we proceed to analyze the experiments at a job (task) level; furthermore, we describe assumptions and simplifications considered for the cleaning of the raw dataset in order to generate a proper dataset.

Section 5 is the main section of the project. In Section 5.1, we review general concepts of *Game Theory* and *Algorithmic Game Theory*, while in Section 5.2, we get into the theory of *Mechanism Design*, from which we get the tools for the formulation of our models. Then, in Section 5.3.1, we present the main model that attempts to capture some of the most important characteristics of the scheduling process that rules *Slurm*, but with the aim of achieving some desired guarantees. We also study a slight variation that, although might appear not very significant, results in interesting results. Moreover, we present a variation of the model in Section 5.4 that attempts to represent the inclusion of a main characteristic of the *Slurm* scheduler. These models are formulated using the previously described theoretical foundations.

In Section 6, we make use of the dataset previously defined and cleaned. We start by briefly describing the sample generation process, along with the justification for the choices we make in this process. Then, in Section 6.2, we describe the experiments that we are going to perform along with the description of the results we evaluate. In Section 6.3.1, we present the results of experimenting with the main model we previously defined, Section 6.3.2 presents the results of experimentation with a slight variation of this model. Then, Section 6.3.3 presents the results for the variation of the main model that tries to consider an important characteristic of the *Slurm* scheduler. These results are accompanied with relevant discussion. Then, in Section 6.4, we arrive to an approximation of the *Price of Anarchy* for the *Slurm* Scheduler, followed by discussion about the *Viability of Cooperation* in Section 6.5.

Appendix C presents plots of relevant distributions of the samples used for experimentation. Appendix B gives a brief description of the experimentation platform developed to run the experiments of the model. In Appendix A, we review the process of setting up the *Slurm* Simulator [1], a process that is not fully described in the original source; we hope this detailed description will be useful for those interested in performing experimentation using this simulator.

Previous work has been done from an algorithmic approach that attempts to directly reduce queuing times among other costs [2][11]. We consider that a formal analysis of the *Slurm* scheduler main mechanics and its variables can lead into a better understanding of this kind of mechanism. On the context of *Algorithmic Game Theory*, there are several studies that deal with scheduling or general competition for resources by a group of players; however, these studies tend to focus more on abstract models, rather than departing from existing mechanisms and their analysis.

As you can infer, this study includes a variety of methods for its development. We start by performing operating systems analysis, through making use of data

analysis tools to construct a proper dataset, then into algorithm and economic theory, followed by some software development, and more data analysis. We hope that this work can showcase in some way the knowledge acquired through the Master in Innovation and Research in Informatics. Nonetheless, these tools and techniques are not only showcased in this study for the sake of it, but they have been necessary for the development of this project.

The code developed and used for this project can be found in the repository `https://earth.bsc.es/gitlab/wuruchi/tfm_agt`; including documentation, data sources, samples, analysis tools, and other relevant resources.

2 Slurm Scheduling Overview

The *Slurm* scheduling mechanism has two main components: Priority and Scheduler.

Priority: A value calculated based on data from the user, and the jobs that the user sends to the High Performance Computer (HPC). The calculation tries to give higher priority to those users that have less usage.

Scheduler: Once the jobs have been received and their Priority calculated, there are certain rules in *Slurm* that determine when a job is sent for execution. As a result, the job with the highest priority is not always executed first, but the order is altered so the resource usage is optimized.

Before we get into more details about these components, it is necessary to give a glimpse of the user hierarchy that *Slurm* manages and uses for its calculations. Also, we get into how and when *Slurm* stores and retrieves the usage information for each user.

2.1 Slurm Users

2.1.1 Users Hierarchy

The users are organized in a hierarchical tree structure, specifically in a Rooted Plane Tree [5], where on top of it we have a main **root** account. Then, the leaves are users and the internal nodes are the accounts to which users are associated. In the *Slurm* documentation, we encounter many references to the term **account**, we consider it equivalent to the term **group**.

Let's describe briefly what we are seeing in Figure 1. The accounts **bsc**, **prace**, and **res** are under the **root** account (we are not listing the sub trees below **prace** and **res**. We have **bsc_es**, **bsc_ls**, and **bsc_cs** as accounts under the account **bsc**; these accounts represent departments of BSC, but there can be users associated directly to **bsc** and to any account at any level. Under **bsc_es** we find two accounts: **bsc30** and **bsc32**; these accounts represent teams inside the BSC Earth Science Department (bsc_es). These teams have users associated to the them, so we have under the account **bsc30**, the users **bsc3090** and **bsc3092**; and under the account **bsc32**, the users **bsc3285**, **bsc3288**, and **bsc3200**. For future reference we will use mainly the sub tree that has as root the account **bsc_es**.

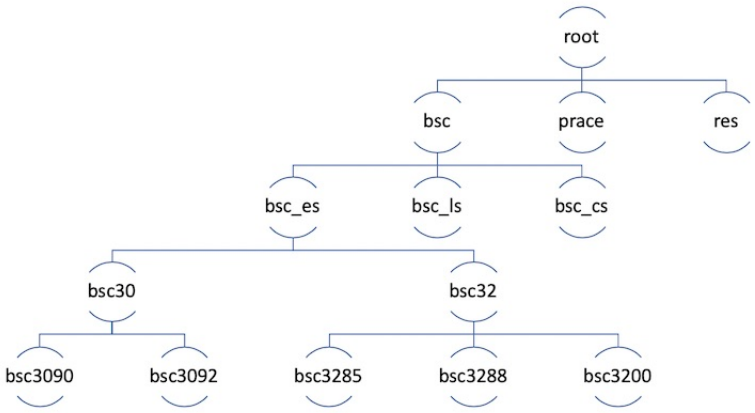

Figure 1: User Hierarchy Example

2.1.2 Usage Definitions

Before we get into **Usage Accounting**, we must describe some relevant definitions.

1. **Shares**: This is an integer value that symbolizes the portion of the computing resource that has been promised to the account or user, this value is set by the administrator of *Slurm*.

2. **Usage**: This is an integer value that symbolizes how much the account or user has consumed from the computing resources.

3. **Usage Unit**: Integer value that is the result of: $CPUs * seconds$, where $CPUs$ is the number of CPUs (or cores) that the job requested, and $seconds$ is the time measured in seconds that the job used from start to finish, not considering time waiting for execution, we can call it **real usage**.

4. **Priority**: Integer value that ranges from 0 to 4294967295. The larger the value, the sooner the job will be scheduled for execution.

5. **Fair-share**: Floating point number between 0.0 and 1.0 that reflects the shares of a computing resource that a user has been allocated and the amount of computing resources the user's jobs have consumed. The higher the value, the higher is the placement in the queue of jobs waiting to be scheduled. For **Fair-share** calculation, the terms **RawShares** and **Shares** are equivalent, **RawUsage** and **Usage** are also equivalent.

We will refer to the sub tree that has as root **bsc_es** in Figure 1, and give an example of how these definitions come into play. The same logic can be applied

User/Account	Shares	Usage
bsc30	3	
bsc3090	1	500
bsc3092	1	1000
bsc32	5	
bsc3285	2	3000
bsc3288	1	3000
bsc3200	1	3000

Table 1: Example of Shares/Usage distribution.

to users and accounts at the **bsc_es** level and upwards. Consider **Shares** and **Usage** values from Table 1.

Account **bsc30** has 3 shares assigned, and **bsc32** has 5. We normalize these values to calculate the portion of the computation resource that corresponds to each account. So, **bsc30** has 37.5% and **bsc32** has 62.5% of the computation resource assigned. We can repeat this calculation for the users under **bsc30**, where each user will have an equal portion assigned (value 1) of the resources corresponding to their account **bsc30**. For the users under **bsc32**, there is a slight difference that, after normalization, means that user **bsc3285** has been assigned 50% of the computation resources assigned to its account **bsc32**, and the rest is divided equally among the other users.

We can apply a similar logic to the usage calculation, having that **bsc3090** has used significantly less resources than **bsc3092** and subsequently will have a higher fair-share factor. At the account level, we have that **bsc32** has used around 86%, which is greater than the shares assigned to this account, as a result, account **bsc30** will have a higher fair-share factor.

2.1.3 Usage Accounting

Slurm stores usage information of currently executing jobs and jobs that have already finished. The procedure that retrieves and stores this information runs in intervals. The length of these intervals is set in the *Slurm* configuration file by the administrator, it is usually a low value so it is possible to know with high probability that the usage returned by *Slurm* reflects the current usage.

Keep in mind that usage is measured in $CPUs \cdot seconds$, where $CPUs$ is the number of cores or CPUs that a job requests for execution. In *MareNostrum4*, there are around 3456 nodes, each node has 2 sockets, each socket has 24 cores or $CPUs$.

2.2 Priority Calculation

The main factors that play in the calculation of the priority are:

1. Age (age_factor)

Weight	Configured Value
$PriorityAgeWeight$	10^5
$PrioritySizeWeight$	10^5
$PriorityFairshareWeight$	10^4
$PriorityQoS$	10^6

Table 2: Example of Weight values configuration.

2. Size ($size\_factor$)
3. Fair-share ($fairshare\_factor$)
4. Quality of Service, commonly known as QoS (qos_factor)

Furthermore, the importance of these factors can be modified by setting weights. Then, the formula for the calculation of priority is:

$$\begin{aligned} Job\_Priority = (PriorityAgeWeight \cdot age\_factor) + \\ (PrioritySizeWeight \cdot size\_factor) + \\ (PriorityFairshareWeight \cdot fairshare\_factor) + \\ (PriorityQoS \cdot qos\_factor) \end{aligned} \tag{1}$$

where $PriorityAgeWeight$ is the weight that modifies the importance of the age_factor in the calculation of Priority. The other weights follow the same logic.

Given that the factors are values between 0.0 and 1.0, it is necessary that these factors take a high value. Suppose that one job has a fair-share factor value of 0.5810 and other has a value of 0.5010, if you pick a $PriorityFairshareWeight$ of 10, the multiplication will result in a not very significant difference. The current configuration of *Slurm* for *Marenostrum4* gives factors the values in Table 2.

The bottom line is that the current *Slurm* configuration gives more importance to the selected **QoS**, then to the **Age** and **Size** of the Job equally, and finally to the **Fair-share**. These three levels differ by 1 order of magnitude.

2.2.1 Age

The longer a job sits in the queue and is eligible to run, the bigger this value gets. This value achieves its maximum value when 10 days have passed. We can say that this value represents how long a job has been waiting for execution. A value from 0.0 to 1.0.

2.2.2 Size

This value is determined by the number of processors ($CPUs$) a job requests. The more processors a job requests the bigger this value gets. This value achieves its maximum value if the job requests all the processors in the HPC. The value is between 0.0 to 1.0.

2.2.3 Fair-share

The *Fair-share* values is determined by Algorithm 1. The algorithm takes as input the number of users, and the representation of the user hierarchy as a Rooted Plane Tree [5], which is a type of the tree data structure. Algorithm 2 is a fundamental function for the calculation of *Fair-share*.

Algorithm 1: Calculate Fairshare by Level Fairshare.

Input: A rooted plane tree: *tree*, that represents the user/account hierarchy. Number of users: n

Output: Ordered list of users according to their Level FS: $list$.

```
root ← tree.root;
rank ← n children ← root.children;
for child in children do
    child.levelFS ← fncLevelFS(child);
end
children_sorted ← sort_by_LevelFS(children);
for child in children_sorted do
    if child == account then
        Recurse with child as root
    else
        child.fairshare ← rank/n;
        rank − −;
        list ← list ∪ {child};
    end
end
```

- $RawShares_{self}$: Shares assigned to this user or account.
- $RawShares_{self+siblings}$: Shares assigned to this user or account and its siblings (users or accounts with the same parent).
- $RawUsage_{self}$: Usage of this user or account.
- $RawUsage_{self+siblings}$: Usage of this user or account and its siblings.

Considering the **bsc_es** sub tree in Figure 1 and Table 1, we calculate **Level FS** and **Fair-share** for each user and account in this example.

Algorithm 2: $fncLevelFS$: Calculate Level FS.

Input: A user or account: *entity*.
Output: Level FS value LF.
$S \leftarrow RawShares_{self}/RawShares_{self+siblings}$;
$U \leftarrow RawUsage_{self}/RawUsage_{self+siblings}$;
if $U == 0$ **then**
  $LF \leftarrow \infty$;
else
  $LF \leftarrow S/U$;
end
return LF;

$fncLevelFS(bsc30) = (3/8)/(1500/10500) = 2.265$
$fncLevelFS(bsc32) = (5/8)/(9000/10500) = 0.729$

Then we proceed to analyze $bsc30$ children first:

$fncLevelFS(bsc3090) = (1/2)/(500/1500) = 1.5$
$bsc3090_{fair-share} = 5/5 = 1$

$fncLevelFS(bsc3092) = (1/2)/(1000/1500) = 0.75$
$bsc3092_{fair-share} = 4/5 = 0.8$

Then, for the children of $bsc32$:

$fncLevelFS(bsc3285) = (2/4)/(3000/9000) = 1.5$
$bsc3285_{fair-share} = 3/5 = 0.6$

$fncLevelFS(bsc3288) = (1/4)/(3000/9000) = 0.75$
$bsc3288_{fair-share} = 2/5 = 0.4$

$fncLevelFS(bsc3200) = (1/4)/(3000/9000) = 0.75$
$bsc3200_{fair-share} = 1/5 = 0.2$

We break ties arbitrarily. Finally, we get the fair-share factor for all the users. User $bsc3090$ will have the highest fair-share factor as it is an under served user.

We can apply this process to the whole account/user tree in Figure 1. In this way, we get the *Fair-share* value for each user that will be used in the calculation of the *Priority* value; subsequently, for the jobs they are going to send, and that will be used to decide the scheduling order.

Name	Priority	MaxWall	MaxCPU	MaxJobs	MaxSubmitJobs
bsc_es	100	2 days	50 nodes	∞	366
debug	10000	2 hours	16 nodes	1	366
interactive	100	2 hours	4 cpus	1	366

Table 3: Example of QoS configuration.

Name	Normalized Priority
bsc_es	0.01
debug	0.98
interactive	0.01

Table 4: Normalized Priority

2.2.4 Quality of Service *QoS*

A *QoS* is an entity that represents a set of rules that apply to the jobs sent using it. These rules impose maximum limits on the number of nodes, processors, or even memory that the jobs can use.

Every *account* and subsequently every *user* under that account is assigned a list of *QoS* that she is allowed to use to submit jobs.

In Table 3, we show an example of priority configuration for a list of *QoS*. There are 48 cores per node, meaning 48 $CPUs$ per node in a standard node. The priority is a value between 0.0 and 1.0; however, we see that it has higher values way beyond the defined boundaries. To achieve a value within the defined range, we normalize the priority values to the highest value. As a result, we have the new values in Table 4.

Using information from Table 3 and 4, we proceed to give a description of the purpose of the *QoS* defined in the implementation of *Slurm* for *MareNostrum4* with the intend of giving intuition about this factor.

The *QoS* **debug** has the highest priority but only 1 job can be running at the time. The jobs sent using this *QoS* cannot run for more than 2 hours, once that limit has been exceeded, the job execution is terminated.

The *QoS* **bsc_es** is the most powerful. It gives the possibility to run jobs that take up to 2 days and up to 50 nodes, also it offers the possibility to have an unlimited number of jobs running at the same time.

The *QoS* **interactive** is the most restrictive, it allows jobs that take up to 2 hours and only 4 $CPUs$; also, only 1 job can be running at the time. The limited amount of $CPUs$ restrict this *QoS* to specific jobs, we can see it as another way of *QoS* design that does not need a high priority because it is going to be used only for specific jobs.

2.3 Scheduling Overview

In Section 2.2, we went through the main variables that play in the calculation of the **Priority** of a job. This is an important value that determines for the

most part the order in which jobs are started. This value effectively produces an execution sequence; however, this ordering can result in sub optimal resource allocation. For example:

Consider a large (in the size of nodes required) job with high priority that is waiting to be scheduled, this job will take 25% of the nodes in the HPC and it has an expected running time of 10 hours; furthermore, it is in the front of the queue. Next to this job, we have a number of smaller jobs requiring a number of nodes from 1% to 2% of the total nodes in the HPC, and expected times lower than 1 hour.

Working under this standard configuration, the scheduler is going to wait for enough resources to be available and then schedule the large job for execution, and this is not optimal because resources will be idle. To avoid this scenario (to some extent), there exists the **backfill** mode. In this mode, the scheduler will start lower priority jobs if that does not delay the start of higher priority jobs. However, this scheduling mode can be a time and computation consuming operation.

For **backfill** mode to work, users should specify a *planned* running time and number of nodes or $CPUs$ for each job they submit. Although they will be charged usage based on the *real* running time, this expected completion time is used for scheduling purposes, as described before. In addition, users are charged based on the number of nodes or $CPUs$ they requested. Since the expected start time of pending jobs depends upon the *planned* completion time of running jobs, it is important that the *planned* completion time is set with accurate values when submitting a job.

The scheduler offers the possibility to impose resource limits on users, but this is already done through the assigned QoS for accounts and users.

In summary, the scheduler considers the order imposed by the Priority value of each job, but ultimately favors resource and time optimization to decide which job is executed next.

These are some relevant configuration variables for the Scheduler mechanism:

- **sched_interval**: Specifies how frequently, in seconds, the main scheduling loop will execute and test all pending jobs.
- **default_queue_depth**: Specifies the number of jobs to consider for scheduling on each event that may result in a job being scheduled. Since the scheduling loop probably repeats often, it is advisable to set a low value for this variable.

These are some relevant configuration variables specific for the **backfill** mode:

- **bf_interval**: Interval between backfill scheduling attempts.
- **bf_max_job_test**: Maximum number of jobs consider for backfill scheduling in each backfill cycle (interval).

- **bf_max_job_user**: Maximum number of jobs to initiate per user in each backfill cycle.
- **bf_window**: How long, in minutes, into the future to look when determining when and where jobs can start.

In Table 5, we have an example of configuration for these variables.

Name	Value	Description
sched_interval	-	Replaced by bf_interval.
default_queue_depth	10000	Considering up to 10000 jobs in queue.
bf_interval	60	Executing backfill scheduling every 60 seconds.
bf_max_job_user	800	Considering only up to 800 jobs per user.
bf_window	10080	Look up to 10080 minutes into the future to detect available resources

Table 5: Example of scheduler configuration settings

3 Scheduling Analysis

In this section, we use the components of the scheduling system previously defined and unify them in a *big picture* presentation using basic mathematical notation in an attempt to make a concise and clear description.

We start by giving a description of the context and main variables involved in the scheduling process in Section 3.1. Section 3.2 presents a somewhat formal description of the scheduling mechanism considering definitions presented in Section 2.

3.1 Scheduling Context

There are M nodes, each of them has a number of CPU_m depending on the type of node, also, the memory can vary too, some nodes have more memory, those are called **fat nodes** and are usually reserved for specific jobs that require that high amount of memory. Furthermore, as an administrative choice, a job i can request k cores (CPUs) where $k < Core_m$ and $Core_m$ is the number of cores of node m, and $Core_m - k$ cores will be available for other jobs. If $k = Core_m$, the whole node is selected. However, if $k > Core_m$ then resources will be assigned for this not on a core basis but in a node basis. For example, if $k = Core_m + 1$ then 2 nodes in their entirety will be assigned for that job. We assume that all nodes M have the same number of $Core_m$, and the same amount of memory, although memory will not be considered. All these M nodes belong to the same **HPC**, i.e. the High Performance Computer. Moreover, as a **simplification choice** we assume that all nodes have the same capacity $Core_i = Core_j \; \forall i \neq j$, along as the same memory, and core speed.

Jobs can request a specific number of cores (processors), according to that number they will be assigned a number of nodes as described in the previous paragraph. More importantly, jobs need to establish a mandatory *planned* running time. Then, we have that a job i has two types of cost w_i as its weight or size, and t_i as the *planned* time in minutes. Also, we have that jobs that have been submitted have an age factor a_i according to how long they have been waiting for scheduling since they have been submitted.

We also consider the collection of QoS that define specific limits for jobs. A user should always select one and only one QoS to sent a job. Each $q \in QoS$ has two types of limits: resource limit (it can be number of cores or number of nodes) rl_q, and time limit tl_q; they also have a priority value $pqos_q$ that can understood as the prize the user gets for using that QoS, given that some of them offer only very limited resources but with a high priority. It is important to distinguish this local priority that only applies to and among QoS from the **Priority** that the scheduler uses.

User $u \in U$ has a share of resources $RawShares_u$, and as resources are consumed, usage is accumulated as $RawUsage_u$. These values will be used in the calculation of the main priority list of the HPC $P_k : N \longrightarrow \{1, ..., n\}$.

So far we have accounted for all the factors involved in the **Priority** calculation as represented by P_i, i.e. the Priority of job i. Next, we proceed to look into the scheduling mechanism.

3.2 Scheduling Mechanism

There are M nodes in the HPC, there are U users that handle experiments. Experiments are represented as *Directed Acyclic Graphs* **DAGs** $G_u = (V_u, E_u)$. For each experiment $G_u \in G$ where G is the set of all experiments from users U, we get a topological order of jobs $V_u^{`} = \{1, 2, ..., n\}$. Users will send job $v_i \in V_u^{`}$ when $\alpha_i = \{v_j | (v_j, v_i) \in E_u \ \wedge status(v_j) \neq 5\} = \emptyset$, where $status()$ is a function that returns 5 if the input job v has status COMPLETED, meaning that all preceding jobs of v_i must have been completed successfully. Then, we define the set of jobs send to the scheduler by user u at a given iteration as $\nu_u = \{v_i | v_i \in V_u^{`} \wedge \alpha_i = \emptyset\}$.

$$N = \bigcup_{u \in U} \nu_u$$

Think of the scheduler as an agent Λ that receives N jobs, each job $i \in N$ has attributes w_i for its weight or size, t_i as the *planned* running time (supplied by the user) in minutes that the job will take to complete, a_i for the time it has been waiting for execution, and p_i for its **Priority** calculated using the previously mentioned attributes. Agent Λ uses the **Priority** p_i as the main ordering principle to define a list $P : N \longrightarrow \{1, ..., n\}$ of execution order.

Agent Λ needs to allocate resources from M to execute N based mainly on P. Usually there will not be enough resources to allocate all $i \in N$, some of these i will have to wait until enough resources are freed for their execution to start, while they wait their attribute a_i increases until a defined max value.

A change in a_i alters p_i; also, execution of $i \in \nu_u$ increases $RawUsage_u$ and subsequently alters p_i. Then, the order in list P will be altered as iterations in the scheduler take place.

Agent Λ not only considers list P to decide which jobs will be allocated resources for execution and in what order, but also considers w_i and t_i to optimize resource usage. For this reason it is important that the user u that sends i specifies accurate values for w_i and t_i.

Regarding w_i, the specified value directly affects p_i through an increase in $RawUsage_u$ directly proportional to w_i. Most $i \in N$ are jobs with high parallelism, which is in itself and advantage given the fact that a factor in the calculation of p_i favors jobs that require more nodes. High parallelism is rewarded, but if u requests more w_i than job i really needs then it should be punished in some way; so far, that way seems to be the increase in $RawUsage_u$ previously mentioned.

Regarding t_i, this value does not affect $RawUsage_u$ because the running time is only measured while n is being executed (RUNNING) and when it has finished execution. It is necessary to reward users that specify values t_i that are closer from above to the measured real execution time because that directly helps the work of agent Λ.

The backfill scheduling mechanism subject of study can work as a **BF_MOD** scheduler as defined in [2]. Under this mode, resources are reserved when a job reaches the top of the queue and is about to be scheduled and the next jobs in the queue can be used for backfilling, but if in the next scheduling event a job with higher priority takes position at the top of the queue, the previous reservation is discarded and a new reservation is executed for the newly arrived top priority job. On the other hand, we have **BF_UNMOD**, where the top job does not change even if a higher priority job arrives, this is also defined in [2]. We assume our scheduler working under **BF_UNMOD** configuration for further discussion.

Remember that we have M machines or nodes of equal capacity, and N jobs sent by U users. For each $i \in N$ we have size w_i, expected execution time t_i, and priority p_i as the main attributes for each job. From now on we make the assumption that w_i is always measured in nodes $1 \leq w_i \leq |M|$ for $w_i \in \mathbb{Z}^+$, t_i in $1 \leq t_n \leq 2880$ (2 days) for $t_i \in \mathbb{Z}^+$, and p_i (as defined in 2.1.2) in $0 \leq p_i \leq 4294967295$ for $p_i \in \mathbb{Z}^+$. Also, the backfill scheduling attempts happen every 1 minute. The system implies a sequence of attempts and job arrivals; however, we will focus on analyzing the system for a single scheduling attempt.

As mentioned before, the backfill scheduling mechanism has been widely studied as an optimization algorithm. It will be considered in the subsequent analysis as a **neutral** mechanism in the definitions of games studied.

4 Data

4.1 Overview

Slurm receives jobs, these jobs come from experiments on which the users are working on. A typical experiment can be modeled as a directed acyclic graph (DAG), it starts with jobs that retrieve or send information, or they might compile software that will be used in the later stages of the experiment. After these initial jobs, there are usually some heavy computation tasks that consist on simulations that implement parallel processes and, subsequently, require many nodes and long running time.

As we mentioned, these experiments can be modeled as a **DAG** $G = (V, E)$ where we have $V = \{1, ..., n\}$ tasks and $V^{`}$ as the list of tasks sorted in topological order with sizes $w_1, w_2, ..., w_n$ measured in the number of **HPC** nodes they require, and $t_1, t_2, ..., t_n$ as the *planned* time in minutes they will need to complete. Typically, a number of vertex at the beginning and end of $V^{`}$ will require less computation resources than the rest in average. We will avoid using the word "node" to name the vertex in a graph to avoid confusion with **HPC** nodes.

Our main source of data is *Autosubmit* [3]. This is a workflow manager for experiments that require the execution of jobs with dependencies between them as previously described. It is implemented as a python library that the handles authentication, job submit, retrieval of job results, among other functions; but more importantly for our current purposes: It stores the the time a job was submitted, started, and finished in a timestamp format. Then, by applying a subtraction, we can get the real execution time of the job. *Autosubmit* allows the user to set the number of processors that the jobs in her experiment will require as an integer value, also, the time the jobs will require for execution in *HH:mm* format, among other variables.

4.2 Data Retrieval

Although *Autosubmit* stores detailed information about the experiments running under it, this information is distributed among different files in a centralized file system. It would have been ideal to have this information in a centralized database to query it easily; as it is not the case we have to develop a script that reduces the complexity of accessing this information and maps it into at most two files:

1. Experiment File: Contains experiment related information.
2. Job File: Contains information specific to the job, retrieving this information is more complex because job specific information is stored in log text files.

Accessing files, reading them, and extracting information is certainly a computation and time consuming operation. To avoid having to wait several hours

every time we need updated information, a **pre-processing worker** has been implemented. This worker acts as a *Map Reduce* agent that crawls through the file system where *Autosubmit* stores the experiments' information, retrieves important information, and updates a centralized database system that stores experiment and job information. Due to time complexity constraints (the worker runs every 5 minutes), not all details about an experiment can be stored, only the most relevant attributes per experiment and job are stored.

Then, our **information retrieval script** will follow these steps:

1. Receive as input the paths to the **Experiment File** and **Job File**.
2. Identify the path to the sources of information: files, folders, database.
3. Verify the existence of these sources of information.
4. Query database for experiment information.
5. Loop through the experiments. In each step, add the experiment information to the **Experiment File**. During this step, it might be necessary to query specific information about the experiment, we use already implemented *Autosubmit* functions to quickly access this data.
6. As a result of the previous step, we identify which experiments are **valid**, meaning those that can provide the information we need. Some experiments could have been altered by users or lack an important file for some reason.
7. Query database for job information.
8. Loop through the jobs that belong to **valid** experiments, in each step add the job information to the **Job File**. It could be necessary to access job specific information in this step, we use *Autosubmit* functions to quickly get this information.
9. Close all connections. End script.

4.3 Description

In Section 4.2, we described the process through which we get the necessary information to start our analysis. In this section, we describe the information we get as a result.

In Table 6, we list the fields of the **Experiment File** where we have highlighted the most important of these. For example, **total** gives us an idea of the complexity of the experiment by the number of jobs that are part of it, this value can range 1 to 50931, which is the experiment with the highest amount of jobs ever registered in *Autosubmit*. However, not every job in an experiment is executed, perhaps the experiment was started and then abandoned at some point, or it just did not start; and that is why we also consider **completed** that gives us the number of jobs that were successfully completed in the experiment.

Field	Type	Description
id	int	Unique internal identifier.
name	string	Unique public identifier.
completed	int	Number of completed jobs.
total	int	Total number of jobs.
user	string	The user owner of the experiment.
as_version	string	*Autosubmit* version.
created	datetime	Date and time when the experiment was created.
model	string	Name of the scientific model used.
hpc	string	Name of the HPC where the experiment runs.
wrapper	string	Type of wrapper (internal *Autosubmit* method).
maxwrapper	int	Number of jobs that can be wrapped.

Table 6: Experiment File

Field	Type	Description
exp_id	int	Unique internal identifier of the experiment.
exp_name	string	Unique public identifier of the experiment.
job_name	string	Unique public identifier of the job.
type	string	Type of the job.
submit	int	Timestamp of the datetime the job was submitted.
start	int	Timestamp of the datetime the job started execution.
finish	int	Timestamp of the datetime the job finished execution.
status	string	Current status of the job.
wallclock	string	Execution time requested in format *HH:mm*.
procs	int	Number of processors requested.
threads	int	Number of threads.
tasks	int	Number of tasks.
queue	string	Name of the queue (*QoS*) targeted.

Table 7: Job File

Using these fields we get an idea of the development of an experiment. More information in Section 4.4.

In Table 7, we describe the fields of the **Job File** where we highlight those fields that we consider specially relevant for this study. A field that deserves immediate mention is **type**, this type is defined when the user configures her experiment in *Autosubmit*, it is to this type that the number of processors and *planned* time requested is associated.

4.4 Analysis

We take a look at the information we have gathered using previously described tools, for this purpose we use *R* under *RStudio*, to easily manipulate the data.

It is important to mention that we are only considering data collected until "30-04-2020", i.e. April 30^{th} 2020.

```
   completed           total                          model              hpc
Min.   :    1.0   Min.   :    2.0   auto-ecearth3         :851   marenostrum4:1114
1st Qu.:   10.0   1st Qu.:   18.0   auto-monarch          :220   marenostrum3: 151
Median :   60.0   Median :   85.0   auto-nemo             : 91   cca          :   0
Mean   :  625.8   Mean   :  726.9   auto-caliope          : 86   cca-intel    :   0
3rd Qu.:  388.0   3rd Qu.:  502.0   ifs_initial_conditions:  6   dl-machine   :   0
Max.   :50931.0   Max.   :50931.0   auto-nmmb             :  5   ecgate       :   0
                                    (Other)               :  6   (Other)      :   0
```

Figure 2: Experiment Data Summary

4.4.1 Experiment Data Analysis

We start by taking a look at the experiments that are usually run in the HPC through *Autosubmit*, so we can get a better understanding of the flow of information in the system that we are analyzing.

Our raw experiment dataset contains much information that is not quite relevant for our study. We proceed to filter some of this information. We are starting with 2734 experiments.

We filter the dataset by the field **hpc** including only those experiments targeting "marenostrum3" or "marenostrum4" because these are the HPC that use *Slurm*. We include only those experiments that have completed at least 10% of their jobs. We should include only experiments that belong to a real experiment, the best way to identify them is by taking the first two characters in the **name** field and if these characters are "t0" then that experiment is a test and should not be considered. Another filter to identify valid experiments is by only considering those where the **model** field is not empty, because valid experiments must have a model defined. Finally, we exclude experiments where the **wrapper** field has any of the values "vertical", "horizontal-vertical", "hybrid", "vertical-mixed", since these wrappers might cause that some jobs have exaggerated queuing times. After we have applied these filters, we have 1265 experiments.

In Figure 2, we take a quick glance at some metrics of our data to get a general idea. We see that we have effectively filtered by the **hpc** field, and we can identify which models are common. Then, by looking at the **total** summary, we can identify that half of our experiments have less than 85 jobs, and the other half has more than 85 up to a maximum of 50931 jobs; also, the **completed** field tells us that most experiments do not complete all their jobs.

We proceed to group the experiments by year and month based on the **created** field. In **R** this can be done by adding a new field where all rows where the datetime field belongs to certain year y and month m are set the datetime value $y - m - 01$ meaning that they are set as if they had the original value set to the first of the month m and year y, for example: "2019-05-01". This is an easy way to group by a datetime field.

In Figure 3, we have accumulated the field **total** by year and month and plotted it in a way that the evolution of the accumulated value across time can be easily appreciated. Notice that the fact that if an experiment is created in a certain month, it does not mean that all its jobs are completed in that month.

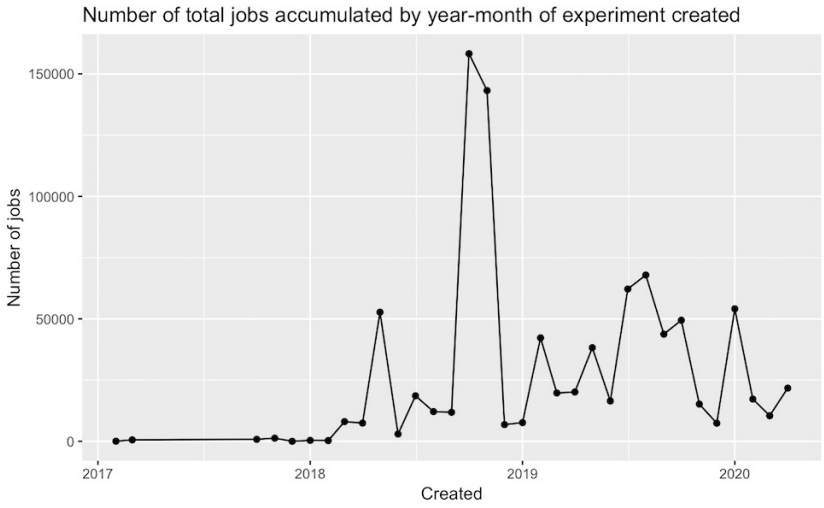

Figure 3: Number of total jobs in experiment by year month

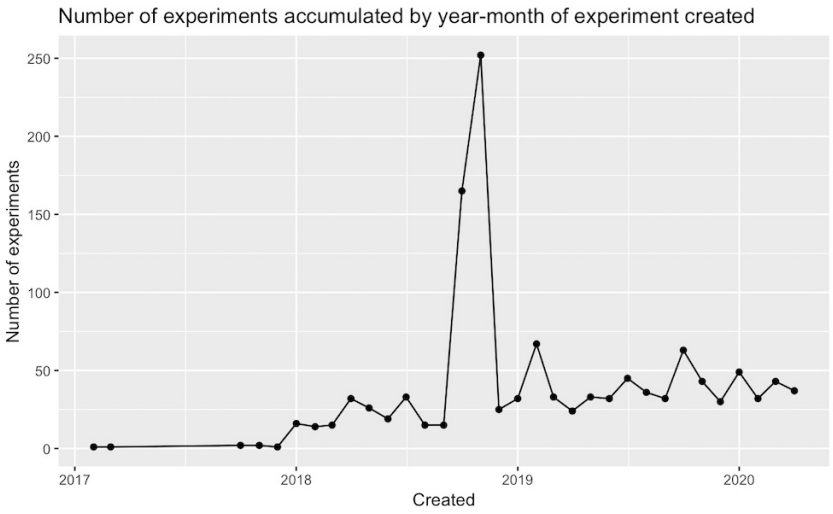

Figure 4: Number of experiments by year month

	model	n_exps
1	auto-ecearth3	850
2	auto-monarch	219
3	auto-nemo	91
4	auto-caliope	86
5	ifs_initial_conditions	6
6	auto-nmmb	5
7	auto-ecearth2	4
8	autosubmit_basic_runtime	2

Table 8: Number of experiments per Model

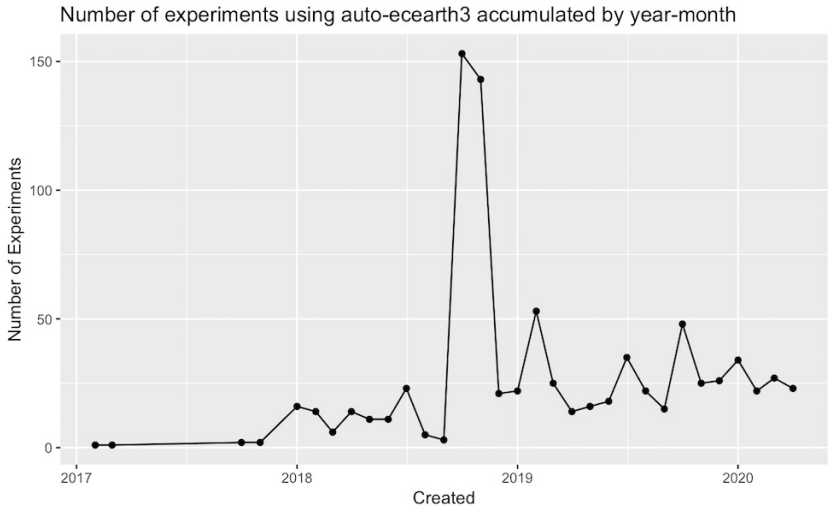

Figure 5: Number of experiments using auto-ecearth3

This is a process that can span several months. Anyway, we are including this plot as a sort of representation of the workload received every month.

In Figure 4, we have accumulated the number of experiments (number of rows) by year and month and plotted it to show the evolution of the accumulated value across time. This plot has the same objective of showing the workload received every month but it can be compared to Figure 3 and we can see that the number of received experiments follows a more regular pattern. There is not much variation for this quantity in the last year.

We have previously mentioned that **model** is an important field that tells us about the validity of an experiment. In Table 8, we count number of experiments per model. This tells us about which models need to be targeted for later experimentation. We start by analyzing the first 4 most used models.

The data for model *auto-ecearth3* plotted in Figure 5 shows a distribution similar to that of Figure 4, which is expected since this is the most used model.

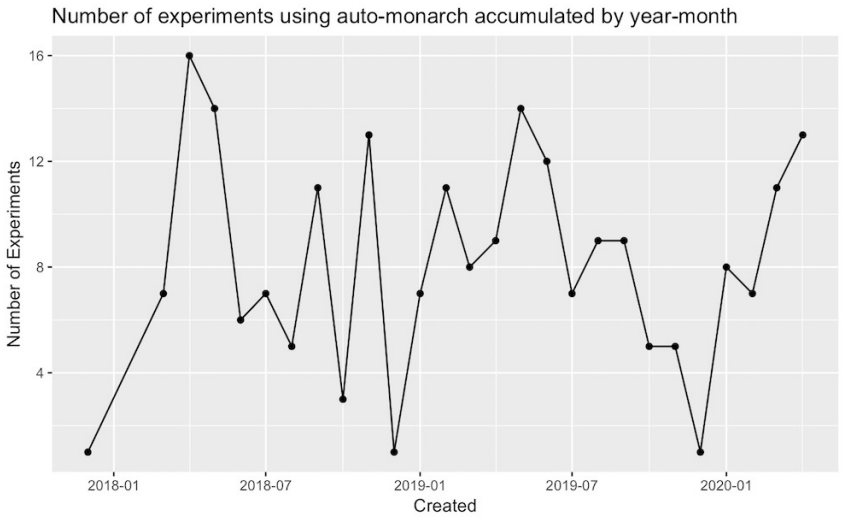

Figure 6: Number of experiments using auto-monarch

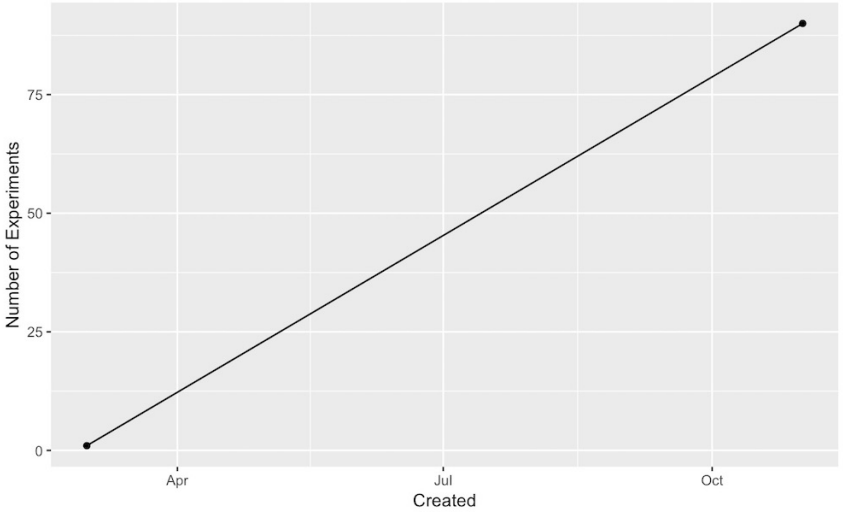

Figure 7: Number of experiments using auto-nemo

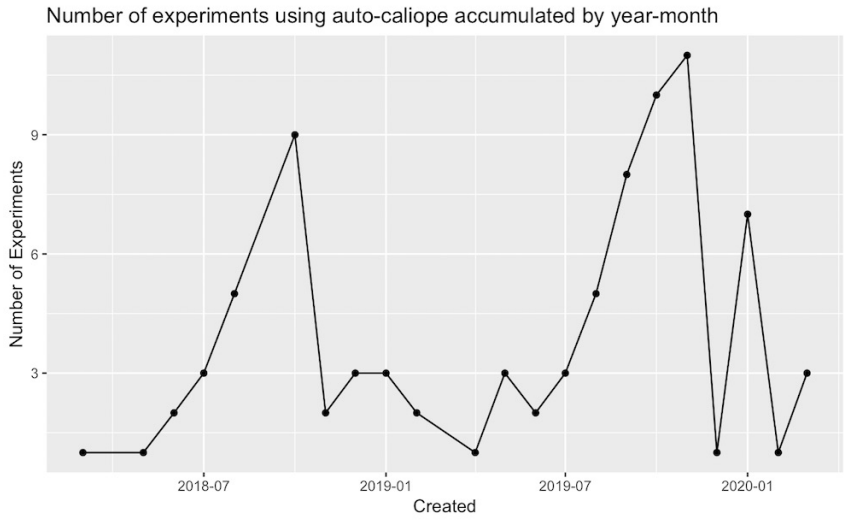

Figure 8: Number of experiments using auto-caliope

QoS	Wallclock *(HH:mm)*
prace	72:00
class_a	72:00
bsc_es	48:00
xlarge	72:00
debug	02:00
interactive	02:00

Table 9: Default Wallclock per *QoS*

Then, in Figure 6 and Figure 8, we see different distributions, but they seem to have been more or less constant for the last two years. In the case of Figure 7, it shows data for the *auto-nemo* model where we see an increase in usage; however, that plot also shows that this model was only used in 2018.

We proceed to filter our dataset to only include experiments belonging to the models *auto-ecearth3*, *auto-monarch*, and *auto-caliope*. This is the subset of 1157 experiments whose jobs will be used in the following analysis.

4.4.2 Job Data Analysis

Before we start analyzing job data, we have to apply some fixes. Some jobs do not have a value for the field **queue** defined, those are assigned to the default *QoS* of the group from which this information is taken from, which corresponds to "*bsc_es*". Also, the important field **wallclock** can be empty for some jobs, in these cases this fields assumes the maximum wallclock value of the *QoS*, see Table 9 for reference.

Our original dataset has 1639409 jobs. First, we filter those jobs that belong to the valid experiment dataset previously defined in Section 4.4.1, this leaves us with 794028 jobs. Then we see that some jobs target specific platforms, this is due to an Autosubmit design decision that gives the user the option to define specific platforms for certain jobs while leaving as default the main platform defined for the experiment. We proceed to include only jobs belonging to platforms we know are related with our target HPC, these are: *marenostrum3*, *marenostrum4*, *marenostrum3-exclusive*, *MareNostrum3*, *marenostrum3_archive*, *marenosturm3*, *marenostrum_archive*, *marenostrum*, *marenostrum4-dt*. After applying this filter, we have 722187 jobs.

Then, we include only those jobs where the field **submit** is greater than 0, meaning a valid timestamp. We only consider jobs with status *COMPLETED* since those are the jobs that will provide enough information for experimentation. Next, we exclude jobs that have the field **queue** with values "interactive", "debug", "DEBUG", "bsc_debug"; these cases represent very restricted *QoS* configurations (see Section 2.2.4) that are not useful for our purposes. In this context, a job (meaning the user) must specify a requested number of processors, if not, we consider it as not valid; so we exclude those jobs that do not specify number of processors in the field **procs**. We know that jobs that request less that 48 processors do not reserve the whole node to which these processors belong, so we decide to exclude jobs requesting less than 48 processors. After these filters have been applied, we end with **102868 jobs that we consider valid jobs**.

We add the fields **minutes** as the value **wallclock** converted to minutes, **nprocs** as the integer representation of **procs**, **nnodes** as **nprocs** divided by 48 (number of processors per node in Marenostrum4) rounding up so we get the number of nodes requested. For our purposes, **nnodes** is the number considered along with **minutes** for resource allocation. We also calculate and add the fields $\mathbf{qtime} = (\mathbf{start} - \mathbf{submit})/60$ and $\mathbf{rtime} = (\mathbf{finish} - \mathbf{start})/60$, which represent the time the job spent in the queue and the time spent running (in minutes) respectively.

At this point we have only jobs that target a *QoS* that allow a wide range of resource allocation, so we can assume that these jobs present an adequate distribution for experimentation. Now we take a look at the distribution of relevant dimensions of this dataset.

In Figure 9, we see information about the most relevant dimensions of our dataset. We have that the **type** *SIM* is the most common, followed by *POST* and *CLEAN*; *SIM* pertains to those jobs that perform the computation of an atmospheric simulation or other type of simulation that usually requires a high amount of computational resources, while the other two types correspond to data handling jobs. We have that the maximum amount of **nodes** requested by a job is 102 and the minimum is 1, where half our jobs request 16 nodes or less. From **minutes** we see that half of our jobs request 40 minutes of computation time or less. We see that **qtime** presents negative values, and the maximum value is too high to be realistic, this indicates that we need to trim some outliers from this dataset. The field **rtime** seems to be more realistic with a maximum

```
      type           nprocs           nnodes
SIM     :92248   Min.   :  48.0   Min.   :  1.00
HERMES  : 3832   1st Qu.: 528.0   1st Qu.: 11.00
WRF     :  896   Median : 768.0   Median : 16.00
CCTM_EU :  837   Mean   : 769.2   Mean   : 16.26
CCTM_IP :  824   3rd Qu.: 768.0   3rd Qu.: 16.00
CCTM_CAT:  615   Max.   :4896.0   Max.   :102.00
(Other) : 3616
   minutes            qtime             rtime
Min.   :   5.00   Min.   :    -32   Min.   : -41.00
1st Qu.:  40.00   1st Qu.:      1   1st Qu.:   4.00
Median :  40.00   Median :      2   Median :  19.00
Mean   :  83.27   Mean   :    139   Mean   :  33.34
3rd Qu.: 120.00   3rd Qu.:     20   3rd Qu.:  32.00
Max.   :3600.00   Max.   :1004533   Max.   :4154.00
```

Figure 9: Job data summary

value of 4154 that stays in the range of allowed running times.

Concerning **qtime**, we start by identifying those rows that contain negative values, 5 jobs. We will not bother trying to find the causes of this misinformation. Since there are only 5 of these cases, we proceed to exclude them. Then, we take a look at Figure 10 where we clearly see extreme outliers. From our knowledge of the HPC and to allow for proper experimentation, we take the decision to include only jobs that queued for at most 12 hours, or 720 minutes. This last filter excludes 2411 jobs.

Regarding **rtime**, we identify 12405 jobs with zero or negative running time. These jobs are not useful for experimentation, so we exclude them. Moreover, jobs where the **rtime** is greater than **minutes** should not be considered, as they break a rule of the HPC and are most likely the result of misinformation. We identify 294 jobs that present this characteristic.

After we apply the filters previously mentioned, we end with **88047 jobs that we consider valid**. This information represents around 5.3% of our original dataset. In Figure 11, we see the summary of the most relevant fields of our valid dataset.

The dataset resulting from the filtering process and described in Figures 12 13 14 15 constitutes the distribution from which we are going to pick samples for experimentation.

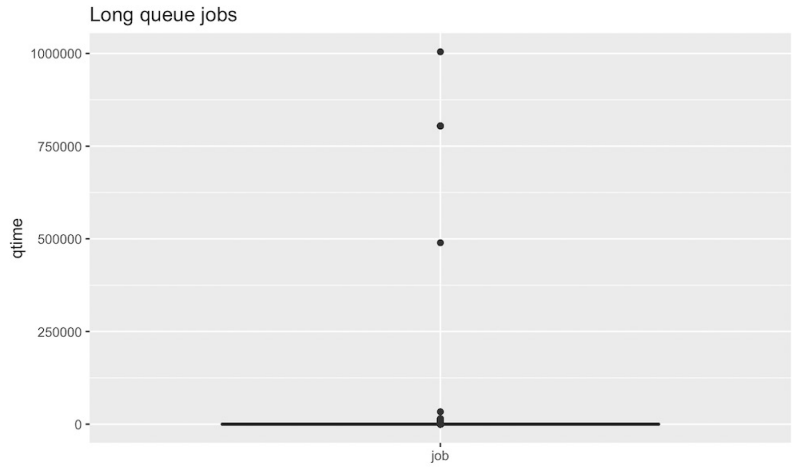

Figure 10: Queuing time per job

```
    nprocs         nnodes
Min.   :  48   Min.   : 1.00
1st Qu.: 672   1st Qu.:14.00
Median : 768   Median :16.00
Mean   : 797   Mean   :16.84
3rd Qu.: 768   3rd Qu.:16.00
Max.   :3456   Max.   :72.00
   minutes           qtime           rtime
Min.   :   5.00   Min.   :  0.00   Min.   :   1.0
1st Qu.:  30.00   1st Qu.:  1.00   1st Qu.:   5.0
Median :  40.00   Median :  2.00   Median :  19.0
Mean   :  75.22   Mean   : 37.38   Mean   :  34.8
3rd Qu.: 120.00   3rd Qu.: 22.00   3rd Qu.:  37.0
Max.   :3600.00   Max.   :720.00   Max.   :1009.0
```

Figure 11: Summary of final job dataset

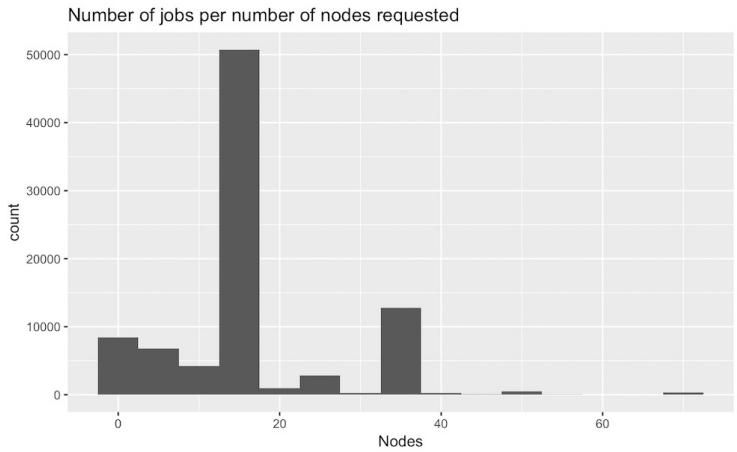

Figure 12: Number of jobs per nodes requested. The bin width has been set to 5 nodes to see more clearly the distribution of the jobs.

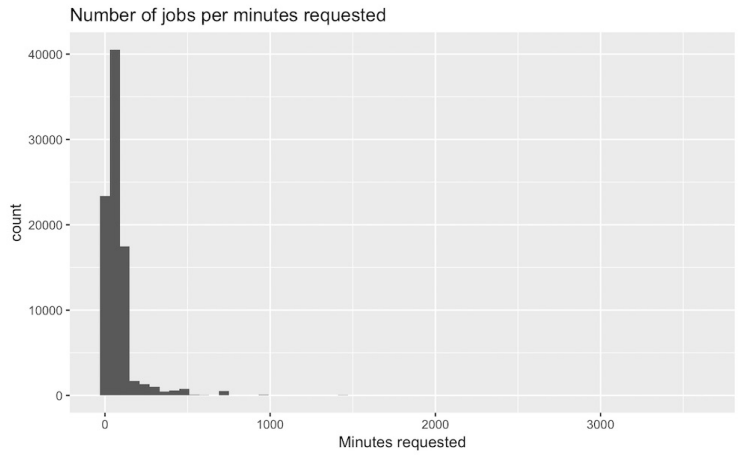

Figure 13: Number of jobs per minutes requested. The bin width has been set to 60 minutes.

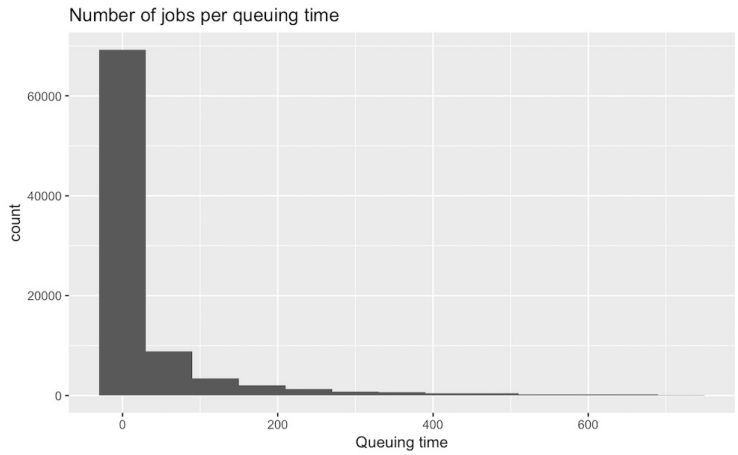

Figure 14: Number of jobs per queuing time recorded. The bin width has been set to 60 minutes.

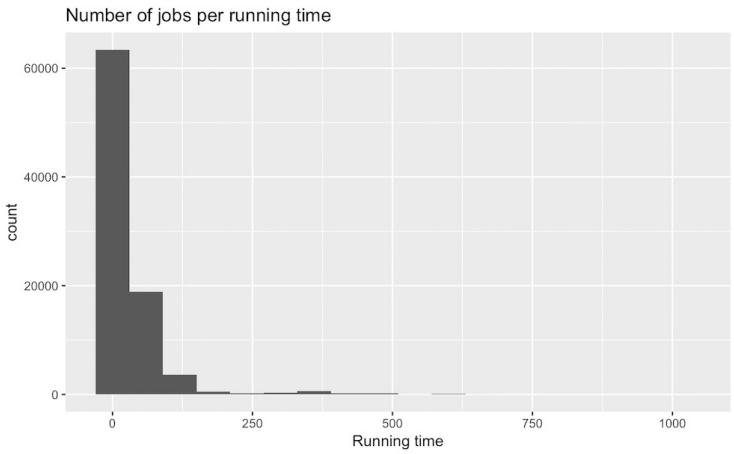

Figure 15: Number of jobs per running time recorded. The bin width has been set to 60 minutes.

5 Games

In previous sections, we have defined the variables involved in the scheduling process along with a description of the main ideas behind the backfill scheduling mechanism. We also reviewed the data that serves as an input to the system, aiming to extract an understanding of the nature of this data. All that information was formalized to get a better grasp of the pieces involved in the functioning of the *Slurm* Scheduler mechanism.

In this section, we start by reviewing the theory that we are going to use to analyze the *Slurm* scheduling mechanism. Then, we propose a model that captures some of the main characteristics of the scheduler into a **Game** where the inherent complexity of the *Slurm* Scheduler is reduced considering assumptions and simplifications.

5.1 Game Theory

5.1.1 Introduction

In the era of the internet, we see that more and more systems involve the participation of many users whose actions influence other users. For example, when users request resources from a single server, by their own actions they increase the load of the server, thus harming any of the other users of the server. In the context of a **High Performance Computer** that we have described in previous sections, we have that users send their jobs for resource allocation and subsequent execution, and by doing this they might influence the resource allocation and execution of the jobs of other users. **Game Theory** aims to model situations in which users/players interact with or influence the outcome of other users/players. The sources of this theory section are mainly the book *Algorithmic Game Theory* [9] and the book by Tim Roughgarden [10].

Most books start their definition of Games with the classic example of **The Prisoner's Dilemma**, but we consider that a more relevant or familiar example for our context is the **ISP Routing Game**, which is another way to express the ideas in the former game.

In Figure 16 [9], we have two Internet Service Providers (ISP) 1 and 2, that need to send traffic from s_1 to t_1 and s_2 to t_2 respectively. The networks corresponding to ISP 1 and 2 exchange traffic via points C and S, which are called *peering points*. Sending traffic through an edge has a cost of 1 for the ISP whose network owns the edge. In this game ISP 1 (player 1) has two strategies: (1) it can send traffic from s_1 to point C incurring a cost of 1 and letting ISP 2 (player 2) handle the cost 4 of sending the traffic to the destination point t_1, (2) or it can send traffic to point S incurring a cost of 2 and letting ISP 2 handle the cost 1 of sending traffic to the destination point. By symmetry, the same applies from ISP 2 perspective. From these two strategies, one is selfish (1), and the other is more considerate towards society.

This is a good moment to look more into **selfish behavior**. We assume that every player tries to get the best result for her, i.e. player is selfish. However,

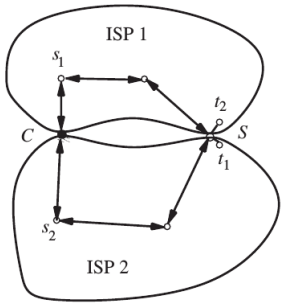

Figure 16: The ISP routing problem.

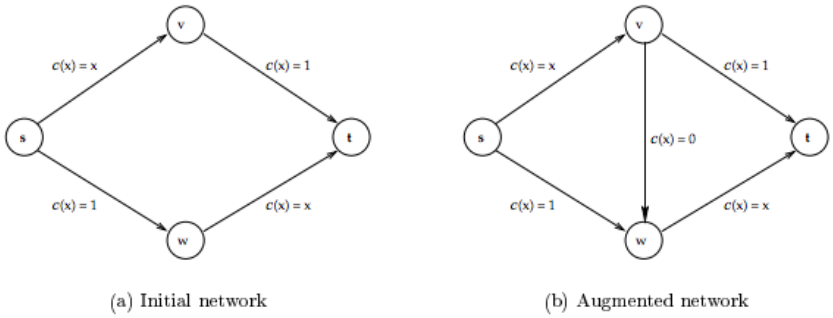

Figure 17: Braess's paradox.

this behavior might result in a worse result than what the player would have got if she did not put her own interests in the first place. An example of the consequences of this behavior is the **Braess's Paradox** [4] that we describe through Figure 17 where $c(x) = x$ means that the delay of going through that route is equal to the fraction of traffic using that route, and $c(x) = 1$ means that the delay of that route is equal to 1. There is a network (a) where there are two routes from s to t, one that goes through v and the other that goes through w. Since both routes are identical, traffic should split evenly between them, achieving a delay of 1.5 for any of the routes. Then, on network (b) a teleportation mechanism is added that takes traffic from v to w without delay. In this new setting, the route $s \longrightarrow v \longrightarrow w \longrightarrow t$ is not worse than the two original routes, but better when some traffic fails to use this route. As a consequence, every driver will selfishly use it and the delay for all drivers will be 2. This delay can be improved to the original value of (a) if some entity distributed traffic evenly between the two original routes. In this example we see that selfish behavior has an undesired consequence from a social standpoint.

We assume that all players act selfishly, and from this "selfish" point of view we see that there is only one optimal strategy for our players, and that is to act selfishly and minimize their costs. In other models of games, selfish behavior could mean that players want to maximize their profit or utility. Imagine that sending traffic from s_1 to t_1 awards some x amount of money to ISP 1, and that sending traffic through an edge has a cost y. Then, the **utility** of ISP 1 would be $x - m * y$ where m is the number of edges. Since x and y are constants, the only way for ISP 1 to **selfishly** maximize its utility is to decrease m. Of course, there are other models of games where the players' selfish optimal strategies depend on the strategies of other players. One example of this is the classic **Tragedy of the commons** where the main idea is that some benefit is achieved only if players carefully choose their selfish strategies in order to reach some objective; otherwise, nobody gets any benefit.

In **The ISP routing game**, it can be argued that a player may get a better benefit by being altruistic and routing traffic through the longer route and expecting the other player to do the same, thus both benefit by cooperation. However, we assume that no player knows what the other is going to do, and must assume that everybody acts **selfishly** and no collaboration is guaranteed.

In games such as **The ISP routing game** we have stable outcomes where each player has selfishly picked an strategy from which she does not want to deviate because it does not give any extra benefit. There are other games where there are no stable outcomes, a classic example is **Matching pennies**. In this game we have two players, each one has a penny and each of them must select a strategy: heads or tails. Player 1 wins if the two pennies match, both heads or both tails, and Player 2 wins if they are different. Player 2 always wants to disrupt Player 1, and Player 1 wants to avoid that disruption. In this example it is best for both Players to randomize their choice of strategy in order to get the maximum expected utility.

5.1.2 Simultaneous Move Game

We will mostly deal with *one-shot simultaneous move games* where all players simultaneously pick their action from their set of the possible strategies. We define an action as the execution of the strategy chosen.

In this type of game we have a set of n players, $\{1, 2, ..., n\}$. Each of these players has a set of possible strategies S_i. To play the game, player i selects a strategy $s_i \in S_i$. Denote by $S = \times_i S_i$ the set of all possible ways in which strategies can be picked. Then, $s \in S$ is the vector of strategies picked by players that determine the outcome of the game for each player. Moreover, between two outcomes S_1 and S_2, we can tell if player i prefers one or the other or regards them as equally good. For example, in the **Matching pennies** game, Player 1 regards outcomes where both pennies match as equally good, so we can state that Player 1 *weakly prefers* S_1 to S_2. We can now try to quantify how much a player prefers one outcome to another, we do this by specifying a utility (also known as payoff) function $u_i : S \longrightarrow \mathbb{R}$ or in other cases a cost function $c_i : S \longrightarrow \mathbb{R}$. Costs and payoffs can be used interchangeably, $u_i = -c_i$. Notice

that in both cases, $u_i(s)$ and $c_i(s)$ consider the set of strategies of all players s, and not just s_i; meaning that the calculation of the outcome of a player must involve the strategies of other players.

5.1.3 Dominant Strategy Solution

In **The ISP routing game** each player has a unique best strategy independent of the strategies of other players, which is to choose the shortest route in their network to minimize costs. Players do not have any incentive to pick any other strategy, as any other option would just mean a higher cost. This best strategy is called a the **dominant strategy**. A game has a *dominant strategy solution* if all players have a dominant strategy.

A strategy $s_i \in S_i$, where S_i is the set of all strategies that player i can play, is a **Dominant Strategy** if player i is at least the same or better off if she adheres to strategy s_i. We represent the strategies played by the other players as s_{-i}. From previous definitions we have that $u_i(s)$ represents the utility of player i for the strategy profile s, so we can use $u(s_i, s_{-i})$ to symbolize the same idea. Then, a strategy profile (also called strategy vector) $s \in S$ is a *dominant strategy solution* if for every player i and any other strategy profile $s' \in S$ we have:

$$u_i(s_i, {s'}_{-i}) \geq u_i({s'}_i, {s'}_{-i})$$

In words, when player i plays s_i, his utility under this strategy while the other players play an alternate profile ${s'}_{-i}$ is at least the same as if she played an alternate strategy ${s'}_i$ while the other players play an alternate profile ${s'}_{-i}$.

A dominant strategy solution for a game does not guarantee optimal utility for any player in a sense that players are not guaranteed to get the best possible utility; although, they can get the best result considering limited information and selfish behavior. For example, as mentioned for the **The ISP routing game**, there is a strategy profile that requires collaboration and improves the utility of both players; however, it is not played by any player because they do not know if the other player is going to collaborate, or if they are going to get betrayed and end up worse. Not every game has a *dominant strategy solution*, in fact, very few have it. The field of **Mechanism Design** establishes the foundations that allow for the design of games with dominant strategy solutions, and even more, that this solution has a socially desirable outcome, or a desirable property for the designer.

5.1.4 Vickrey Auction

An auction presents a situation in which we can clearly identify the elements of a game, the situation is even more clear for a single item auction. In this auction we have players (bidders) and each player i has a private value v_i for the item, i.e. what this player is willing to spend for the item. The strategy for each player is her bid, if player i wins the auction she gets utility $p - v_i$, where p is the price at which the item is finally sold; if player i does not win, her utility

is 0. We consider a *one-shot simultaneous move game* where all players submit their bids at the same time or if they submit their bids privately to a neutral agent. At first sight, it does not seem clear which should be the bid of player i, certainly it should be some number greater than the second highest bid, but player i does not know the value of the second highest bid.

In the Vickrey's mechanism, also known as the *Second Price Auction*, the item is awarded to the highest bidder, but that bidder pays an amount equal to the second highest bid. Turns out that this mechanism has the property that the **dominant strategy** of each player is simply to bid his true value, even without knowing what other players bid; or said otherwise, the dominant strategy is to tell the truth.

This mechanism has three important properties: It is welfare maximizing, meaning that the item is awarded to the person that values it the most; it is easy for the players to play, as they only have to play their dominant strategies; finally, considering the properties previously listed, it is fairly simple to calculate who gets the item and how much this player must pay, so we can say that this mechanism is "computationally efficient",

5.1.5 Pure Strategy Nash Equilibrium

A second price auction is a special case where there exists a dominant strategy solution, most games do not have this property. So we need a more broadly applicable solution concept, and that is the **Nash Equilibrium**. In a Nash Equilibrium all players act selfishly in order to maximize their utilities, and this behavior results in a stable solution. A strategy profile $s \in S$ is a *Pure Strategy Nash equilibrium* if for all players i, and for and any alternate profile $s'_i \in S_i$, we have:

$$u_i(s_i, s_{-i}) \geq u_i(s'_i, s_{-i})$$

Meaning that under strategy profile s no player i can pick an alternate strategy $s'_i \in S_i$ while other players continue to adhere to s_{-i} and thus increase her utility. Notice the difference with the definition of *dominant strategy*, where other players do not need to adhere to s_{-i} but pick from any other $s' \in S$. Having said that, it is evident that any dominant strategy solution is a Nash equilibrium, and if switching to this dominant strategy always improves utility, then it is the unique Nash equilibrium of the game. However, consider that a game can have multiple equilibria, and that different equilibria can have a different utility for the players.

Having a game with possible multiple equilibria and selfish independent players makes it hard to predict what players should do, or what is going to happen in the outcome of the game. The important characteristic here is that once a Nash equilibrium is reached, players will not want to individually deviate.

5.1.6 Mixed Strategy Nash Equilibria

As mentioned at the end of Section 5.1.1, we have games where there are no pure strategy Nash equilibria, meaning that there is no stable solution that involves players deterministically picking a strategy and playing it; instead it is best for them to randomize between the strategies that aim to maximize the *expected payoff*. Adding to the definition in Section 5.1.5, each player now picks a probability distribution over her possible strategies, we call this distribution a *Mixed Strategy*. Then they select an action from this distribution and play it.

Considering *mixed strategies*, Nash [8] proved that every game with a finite number of players, and each of them with a finite number of strategies, has a Nash Equilibrium.

5.1.7 The Price of Anarchy

The Price of Anarchy is the most popular way to measure the inefficiency of equilibria in games with multiple equilibria. This metric is defined as the ratio between the worst value of the objective function of an equilibrium and the optimal value of the same objective function.

Consider a game $G = (N, S, u)$ defined by a set of players N, strategies S_i for each player i, and a utility function $u_i : S \longrightarrow \mathbb{R}$. We also define a welfare function $Welf : S \longrightarrow \mathbb{R}$ calculated by the sum of players utilities $Welf(s) = \sum_{i \in N} u_i(s)$, where $s \in Equil$ is a strategy profile in equilibrium, and $Equil \subseteq S$.

$$PoA = \frac{max_{s \in S} Welf(s)}{min_{s \in Equil} Welf(s)}$$

If instead we chose to minimize a cost function $Cost : S \longrightarrow \mathbb{R}$, we define **PoA** as:

$$PoA = \frac{max_{s \in Equil} Cost(s)}{min_{s \in S} Cost(s)}$$

Notice that this metric is defined based on the choice of objective function. This is a key idea, since our argument about this metric in Section 6.4 is based on our particular choice of objective function.

5.2 Mechanism Design

Mechanism Design is the science of rule-making. This sub-field of economic theory, which has an engineering perspective, is interested in designing economic mechanisms that achieve a socially desirable outcome, or a desirable property defined by the designer. We define this social outcome as the aggregation of the preferences of the participants in the mechanism into a single joint decision, we call this outcome a *social choice*. We assume that players in this mechanism act rationally in an economic sense, which implies selfishness.

5.2.1 Introduction

In Section 5.1.4, we saw that the **Vickrey Auction** enjoys three desirable properties:

1. **Strong incentive guarantees:** Truthful bidding is a dominant strategy and **never leads to a negative utility**. We refer to auctions that comply with this guarantee as *Dominant Strategy Incentive Compatible* (**DSIC**). An auction with this guarantee is easy to play for bidders, and assuming that the bidders play their dominant strategies, we can predict the outcome of the auction.

2. **Strong performance guarantees:** Social welfare maximization, the bidder with the highest value gets the item, assuming truthful bids. The outcome of the auction maximizes $\sum_{i=1}^{n} v_i \cdot x_i$ where x_i is the amount of items allocated to bidder i, in this case it is 1 for the winner and 0 for the others.

3. **Computationally efficient:** The auction can be implemented in polynomial time, linear for this case.

We refer to the Vickrey auction as an ideal auction, and we will refer to any mechanism/auction that has the 3 properties mentioned as an ideal mechanism/auction.

5.2.2 Design Approach

The properties in Section 3.2 are well defined for Vickrey Auctions, but we want to extend them to more general problems. To achieve this we follow a 2-step approach:

- **Step 1:** Assume that bidders bid truthfully. How should we allocate item(s) to bidder(s) so the properties **Strong performance guarantee** and **Computationally efficient** hold?

- **Step 2:** Given the allocation rule defined in **Step 1**, how should we set the selling prices so **Strong incentive guarantees** holds? Meaning that our payment rule should promote dominant strategies, and subsequently, a DSIC mechanism.

In general, an **allocation rule** defines which bidder gets which item, and a **payment rule** defines the payment that bidders make to the mechanism. We define these concepts more formally later, but it was necessary to mention them in this Section as to make clear the connection between the design approach and the next concepts. Furthermore, *Myerson's Lemma* is a general and powerful tool for implementing Step 2, the payment rule.

5.2.3 Single-Parameter Environments

This is the level of abstraction at which we are going to set our analysis, and at which to state *Myerson's Lemma*. In this environment there are a number n of players (or bidders), each player i has a private valuation v_i per item. There is a feasible set X such that each element of this set is an n-vector $(x_1, x_2, ..., x_n)$ where x_i is the amount of items player i gets. For example, in a single-item auction (an auction where only 1 item is auctioned among n bidders) each element of X is a 0-1 vector such that $\sum_{i=1}^{n} x_i \leq 1$. In this setting the valuation v_i for when player i wins is a **single value** v_i, and 0 if she loses. On the other hand, consider the auction where several different items are auctioned at the same time, in this case there is no single v_i for when player i wins some set of items because the player might have preferences between the items resulting in a different v_i for a different winning outcome. We call this a **multi-parameter** environment.

5.2.4 Allocation and Payment Rules

In Section 5.2.2, we saw that our mechanism needs to make two important choices: allocate items and define payments. In the auction context, these decisions are formalized as an *allocation rule* to define who receives what, and a *payment rule* that defines how much the winning players has to pay to the mechanism. The steps to apply these rules are:

1. Collect bids $b = (b_1, ..., b_n)$ from all bidders (players). Vector b is the bid vector or bid profile.

2. Through the **allocation rule**: Choose a feasible allocation $\mathbf{x(b)} \in X \subseteq \mathbb{R}^n$ as a function of the bid vector.

3. Through the **payment rule**: Choose payments $\mathbf{p(b)} \in \mathbb{R}^n$ as a function of the bid vector.

The procedure defined in the previous steps is called a *direct-revelation mechanism* because the players reveal their private information in the first step. There is another type of procedure called *indirect revelation mechanism* where the private information revelation is not required, an example is an iterative ascending auction, also known as *English Auction*.

We use a *quasilinear utility function* for our model, where we have that with allocation rule $\mathbf{x}$ and payment rule $\mathbf{p}$, player i has utility:

$$u_i(\mathbf{b}) = v_i \cdot x_i(\mathbf{b}) - p_i(\mathbf{b})$$

5.2.5 Monotonicity

In general we have that an **allocation rule** $\mathbf{x}$, also known as a social choice function, satisfies *Weak Monotonicity* if for all players i and all v_{-i}, we have that $\mathbf{x}(v_i, v_{-i}) = a \neq b = \mathbf{x}(v', v_{-i})$ implies that $v_i(a) - v_i(b) \geq v'_i(a) -$

$v'_i(b)$. Meaning that if the social choice changes when a single player changes his valuations, then it must be because this player increased his value of the new choice compared to that of the old choice.

Then, for **single-parameter environments** we have that an **allocation rule x**, also known as a social choice function, is called *monotone* in v_i if for every v_{-i} and every $v_i \leq v'_i \in \mathbb{R}$ we have that $\mathbf{x}(v_i, v_{-i}) \in W_i$ implies that $\mathbf{x}(v'_i, v_{-i}) \in W_i$, where values are chosen from a subset of winning alternatives $W_i \subseteq A$, and A is the set of alternatives. In other words, if value v_i makes a player i win, then bidding $v'_i > v_i$ will also make player i win.

5.2.6 Myerson's Lemma

We have two important definitions:

1. **Implementable Allocation Rule**: An allocation rule $\mathbf{x}$ for a single-parameter environment is *implementable* if there is a payment rule $\mathbf{p}$ such that the direct-revelation mechanism ($\mathbf{x}$,$\mathbf{p}$) is *DSIC*. Meaning that implementable allocation rules are those that extend to *DSIC* mechanisms. This definition is important because we have that **Ideal Auctions** as defined in Section 5.2.1 must be *DSIC*.

2. **Monotone Allocation Rule:** An allocation rule $\mathbf{x}$ for a single-parameter environment is *monotone* if for every player i and bids $\mathbf{b}_{-i}$ by the other players, the allocation $x_i(z, \mathbf{b}_{-i})$ to i is non-decreasing in her bid z. Meaning that bidding higher can only get you more items, if there are more items to win, or continue to win in case there is only 1 item and you were already winning. **An allocation rule must be monotone to be implementable**.

We state **Myerson's Lemma**:

Theorem 1 ***Myerson's Lemma*** *Fix a single-paramenter environment.*

(a) An allocation rule $\boldsymbol{x}$ *is implementable if an only if it is* ***monotone****.*

(b) If $\boldsymbol{x}$ *is monotone, then there is a unique payment rule for which the direct-revelation mechanism* ($\boldsymbol{x}$,$\boldsymbol{p}$) *is DSIC and* $p_i(\boldsymbol{b}) = 0$ *whenever* $b_i = 0$*.*

(c) They payment rule is given by an explicit formula.

Part **(a)** states that previous definitions 1 and 2 define the same class of allocation rules, where definition 2 is the more operational as it is not hard to check whether an allocation is monotone or not. Part **(b)** states that if an allocation rule is implementable then it is straightforward how to set payments to achieve a the *DSIC* property; moreover, there is only one way to do it.

We can summarize the most relevant, for our purposes, part of this theorem as: If a direct-revelation mechanism $(\mathbf{x}, \mathbf{p})$ is *DSIC*, then the allocation rule $\mathbf{x}$ satisfies monotonicity. If there is a monotone allocation rule $\mathbf{x}$, then there is a payment rule $\mathbf{p}$ such that $(\mathbf{x}, \mathbf{p})$ is *DSIC*.

5.2.7 Vickrey-Clark-Groves Auction

This is a generalization for **multiple items** of the *Vickrey second price auction* seen in Section 5.1.4. In this general setting we have n players bidding for A items, players can submit more than one bid, since they might have different valuations for different amount of items received. This is a *direct-revelation* setting where each player's bids are private. Then, the mechanism considers all combinations of the bids and picks the combination that maximizes social welfare $\mathbf{x}(b) \in argmax_{a \in A} \sum_i v_i(a)$, for an allocation rule $\mathbf{x}$. Then, the players whose bids were chosen by the mechanism receive their items and pay not the amount they had bid, but only the marginal harm their bid has caused to the other players, by making them win less items or none at all, where this payment can be as high as their original bid, otherwise *DSIC* is lost because players get negative utility.

With rational (in the economic sense) players, this mechanism is *DSIC* and the dominant strategy for players is to be truthful. Then, we have that:

(a) The only incentive compatible mechanisms that maximize *Social Welfare* are those with VCG payments.

(b) In a single-item auction, the amount the wining player pays for the awarded item is equal to the value of the winner in the case he had not participated in the auction minus the value of the other players in case he participates in the auction. Given that in the case where he participates in the auction the value of the losing players is 0, the VCG payment rule says that the winning player should play the second highest bid (or value, assuming truthful bids), which is the same payment rule as in the **ideal** Vickrey second price auction.

5.2.8 Knapsack Auction

Knapsack Auctions are another example of single-parameter environments. In a *Knapsack Auction* each player has a publicly known size w_i and a privately known valuation. The system has capacity W. The feasible set X is the 0-1 n-vector $(x_1, x_2, ..., x_n)$ such that, $\sum_i w_i x_i \leq W$, where $x_i = 1$ if i is a winning bidder.

In Section 5.2.2, we defined a 2-step approach to design a *DSIC* mechanism. **First**, we design an allocation rule that maximizes welfare:

$$\mathbf{x(b)} = argmax_X \sum_{i=1}^{n} b_i x_i$$

The allocation rule solves an instance of the knapsack problem where the item values are the reported bids $b_1, b_2, ..., b_n$ and the item sizes are the known sizes $w_1, w_2, ..., w_n$. When players bid truthfully, this allocation rule maximizes social welfare. Moreover, this allocation rule is **monotone**.

Remember that an instance of a knapsack problems consists of $2n+1$ positive numbers. We have the values of the items (the bids in this context) $b_1, b_2, ..., b_n$,

the sizes of the items are $w_1, w_2, ..., w_n$, and the capacity of the knapsack W. The objective is to compute the subset of items of maximum total value that have total size at most W.

Secondly, we define a payment rule that extends the allocation rule to a *DSIC* the mechanism. Myerson's lemma, see Section 5.2.6 guarantees the existence of a payment rule $\mathbf{p}$ such that the mechanism $(\mathbf{x}, \mathbf{p})$ is *DSIC*. Under this payment rule $\mathbf{p}$, a bidder i that wins pays her **critical bid**, which is the lowest bid she can make and continue to win, fixing the other bids b_{-i}. This is analogous to the payment rule of the *Vickrey Second Price Auction*.

We have already shown that this mechanism aims at welfare maximization, but to be ideal it must have the properties described in Section 5.2.1: *DSIC*, Welfare Maximizing, and **Computationally Efficient**. Our allocation rule should give a result in polynomial time as a function of the input to be considered computationally efficient, even better is linear time. We know that the Knapsack problem is *NP-hard*. This means that there is no polynomial time algorithm that implements the allocation rule in polynomial time as a function of its input. However, we can use other methods to attain the required efficiency; for example, **approximation algorithms**.

There is a similarity between *Mechanism Design* and the field of *Approximation Algorithms* in that both have the primary goal of designing polynomial time algorithms for difficult problems, but with the difference that *Mechanism Design* also adds an additional monotonicity constraint.

5.2.9 Greedy Knapsack Algorithm

In this section, we study one of the many heuristics that attempt to solve the Knapsack problem with good worst-case performance guarantee.

Consider a bid profile $\mathbf{b}$ and a set of winners I with total size $\sum_{i \in I} w_i \leq W$. For all $i \in I$, we have that $w_i \leq W$. Then we follow the steps:

1. Sort and re-index the bidders so that:
$$\frac{b_1}{w_1} \geq \frac{b_2}{w_2} \geq \cdots \geq \frac{b_n}{w_n}$$
2. Pick items in that order until one does not fit and halt.
3. Return the solution from the previous step or the highest bidder, whichever has largest social welfare.

Assuming truthful bids, the social welfare achieved by the greedy algorithm is at least 50% of the maximum social welfare. Furthermore, this algorithm can achieve even better results if $m_i \leq \beta \cdot W$ for every player i and $\beta \in (0, \frac{1}{2}]$, then the approximation guarantee increases to $1 - \beta$.

5.3 Scheduler Model

The *Slurm* Scheduler is a complex mechanism that involves several variables form different sources, some of them depend on the user, others depend on the users that share a group with the user, others depend on configurations settings. These variables can also depend on the specific project on which the users are working on. In this context, it is difficult to perform an analysis of the system as a single entity; so instead, we take a look at some of the characteristics of the *Slurm* Scheduler that can be used in an strategic way by the users. We isolate these characteristics in an independent environment so we can model them and get a clear understanding of the mechanics at play, in an attempt to get a better understanding of the system.

We measure performance as the maximization of an objective function. We define our objective function as the **Total Value** achieved by the scheduler, which is the sum of the values each user has for the execution of her job.

5.3.1 Knapsack Auction Model

We model the *Slurm* Scheduler as a *Knapsack Auction*, which is described in Section 5.2.8. We have seen that the Knapsack Auction model has some desired properties that will result in potential optimal results that we can analyze and use for comparison purposes. We will study some variables used by the *Slurm* Scheduler that can be adapted to the Knapsack mechanism without making this model lose its important properties.

One of the main variables in the *Slurm* Scheduler is **Priority**, this value is calculated based on factors that try to capture as much information as possible from the environment while also trying to preserve some kind of *fairness*. This value directly affects how jobs are considered for execution.

In this model, we ignore most of the factors that are used for the calculation of Priority. This can be seen as starting the system from scratch, where the Priority calculation described in Section 2.2 is as simple as it gets. We can state this environment as if there is no data that conditions the calculation of the Priority.

However, we are not going to ignore Priority completely, since we will still consider the size and running time (*real* and *planned*) of the job that is sent to the scheduler, as well as the concept of *user*. However, for our model we will not consider the hierarchy of users, but a single group to which all users belong. This can also be seen as if we analyze a single group of the hierarchy independently, where that group has some reservation of resources, number of nodes, in which they can run their jobs.

We have a set of n players that submit n jobs, 1 job per player. Player i has available some currency that she is going to use to pay for the resources her job i consumes. In the context of *Slurm*, this currency can be thought as another factor in the *Priority calculation* formula. Furthermore, this currency concept has some resemblance to the concept of *Usage*, where the user pays with *Usage* the execution of her jobs. As a consequence, later jobs receive a penalty in the

calculation of Priority by the *Slurm* Scheduler.

Player i submits job i that has a known size w_i representing the number of nodes that job i requires, and t_i representing the number of minutes that job i requires to finish execution (*planned* running time). Considering w_i and t_i, player i has a privately known value v_i that she gives to the execution of job i. In the Knapsack Auction, the values v_i of each player are represented by their reported bids $b_1, b_2, ..., b_n$. As we have seen in the definition of the Knapsack Auction, the best strategy for a player is to report her truthful value, meaning $b_i = v_i$. So we consider this assumption in this model as well.

Consider that when job i is allocated resources for execution, it will run on the specified number of nodes w_i for as long as the specified *planned* running time t_i. When that time has been consumed, the job is terminated independently if it has been completed or not. Player i is aware of this fact, subsequently, she will not send a value t_i lower than the expected required time of her job; also, to secure the completion of her job, player i might add some extra amount of time to cover for unexpected variations, some ϵ.

Consider the idea of a computation resource that we measure as $nodes \times time$. In our context, we measure this computation resources in units of $nodes \times minutes$ or equivalently as $w_i \cdot t_i$ for job i of player i.

Our allocation rule $\mathbf{x}$ should *try* to maximize *Social Welfare* and do it in a *computationally efficient* way. To achieve this objective, we are going to use the greedy algorithm defined in Section 5.2.9.

1. Sort and re-index the bidders so that:

$$\frac{b_1}{w_1} \geq \frac{b_2}{w_2} \geq \cdots \geq \frac{b_n}{w_n}$$

2. Pick items in that order until one does not fit and halt.

3. Return the solution from the previous step or the highest bidder, whichever has largest social welfare.

We maintain that $\sum_i w_i \leq W$, for the subset of winning jobs/players. The original greedy algorithm considered only $\frac{b_i}{w_i}$. In our setting, it is tempting to also consider the time these w_i resources are going to be unavailable because they are allocated to job i, to include this idea we could use $\frac{b_i}{w_i \cdot rtime_i}$. Where $rtime_i$ symbolizes the idea of *expected* running time, meaning that based on previous runs of similar jobs, the scheduler can calculate an expectation $rtime_i$ of the *real* time this job is going to be running for. We will see more about the effect of including the $rtime_i$ into the greedy algorithm in Section 6; where we take $rtime_i$ not as an *expected* value, but as the *real* running time of the job. This simplification is made because we can get this value from our datasource, and we are testing a *single attempt* model, where there are no previous jobs from which to get an average.

Another possible improvement could be to alter step 2, so the algorithm does not stop when it finds a job that does not fit current capacity but keeps going

until it finds a job that fits the remaining capacity or it reaches the end of the list. However, this might result in breaking the monotonicity of the allocation rule, thus breaking *DSIC*. We will explore this scenario in Section 6.3.2.

This greedy algorithm allocation rule is monotonic, but we certainly get a greater or equal allocation guarantee by using a exact optimal solution algorithm. In Section 6, we also test our samples using a branch and bound based algorithm implemented in Google OR-Tools [6] that finds an optimal solution. However, due to our choice of b_i and the distribution of our datasource, our optimal solver might take too long to arrive to a solution.

Our payment rule $\mathbf{p}$ should extend the allocation rule $\mathbf{x}$ to *DSIC*. For this purpose we use the VCG Auction payment rule, defined in Section 5.2.7, in which a winning player pays the marginal harm her bid inflicts to the other players, such marginal harm can be as high as her original bid.

$$\mathbf{p}_i(\mathbf{b}) = argmax_z \sum_{i \neq j} b_j(z) - \sum_{i \neq j} b_j(a)$$

Considering $a = \mathbf{x}(\mathbf{b})$ as the outcome that includes player i, and $z = \mathbf{x}(\mathbf{b}_{-i})$ as the outcome of the greedy algorithm where i did not participate. In this way, we calculate the payment of player i as the difference between the result of the chosen algorithm or solver when i does not participate and the result when i participates but not aggregating bid b_i. The result is a payment that represents the harm imposed by player i on the other players by participating, or also mentioned in textbooks as **making the player internalize her externalities**.

It makes sense to consider a VCG payment rule since we are not trying to maximize revenue but social welfare. One of the main criticisms of the VCG payment rule is that it is not designed to maximize revenue, thus it is not widely applied in practice, but in this case we care only about maximizing social welfare.

Notice that to calculate the VCG payments, our algorithm has to calculate the solution for $k+1$ instances of the Knapsack problem, where k is the number of jobs included in the Knapsack.

Does a VCG payment rule extends our allocation rule to a DSIC mechanism? We will show it through experimentation results in Section 6.3.

5.4 Knapsack Auction Model + Priority

We have designed a model that tries to capture some important characteristics of the *Slurm* Scheduler. This model has some strong performance guarantees based on the information provided by the user. However, the *Slurm* Scheduler implements an important factor that we have disregarded so far, **Priority**.

If we consider **Priority** as the deciding factor for the allocation rule, then, our allocation rule is no longer monotone. As a consequence, we can conclude that *Social Welfare* is no longer maximized.

In this model, we no longer use the greedy or optimal algorithm to find an ordering and select the jobs that will be included in the Knapsack. Instead, we use a **Priority** value to calculate the ordering. We test this scenario in Section 6.3.3.

6 Experimentation and Results

In Section 4.4, we analyzed data from a workflow management system that runs jobs in the HPC. In this section, we will use this data to generate samples from this data. Then, we perform experimentation using these samples.

We have mentioned before that *MareNostrum4*, the HPC that uses *Slurm* as its scheduler, has 3456 computation nodes. This number will be key in our experimentation, as we will see in the following sections.

6.1 Sample Generation

We proceed to take the output of the analysis performed in Section 4.4 and produce random samples. We consider that our HPC has 3456 nodes. The data we have sums 1482526 nodes, which is a very large number that might be distributed across several days. We will assume that at any given moment the HPC receives around twice the amount of resource requirement than it can handle; although, according to *MareNostrum4* management, this factor is closer to 1.2. For experimentation purposes, we will choose jobs uniformly at random until they sum no less than 6912 nodes. There will be cases where the last included job makes the sum go over 6912, we still consider it and halt the sampling procedure.

These samples are then saved in a text format for later usage. For each job in the sample we save the attributes:

1. **job_name:** A unique string identifier.
2. **type:** Type of job, might be relevant.
3. **nnodes:** Number of nodes the job requests w_i.
4. **qtime:** Queuing time in minutes the job spent in the queue of the scheduler.
5. **rtime:** Running time the job took until completion.

6.2 Experimentation Space

Consider the values:

1. W: Total number of nodes in the HPC (3456).
2. n : Number of players (jobs).
3. w_i : Weight of job i measured in nodes.
4. b_i : Bid of player i for job i.
5. p_i : Payment of player i for job i.
6. d_i : $\sqrt{(b_i - p_i)^2}$.

7. x_i : 1 if job i is selected into the knapsack, 0 otherwise.

8. min_i : Planned time in minutes that player i considers job i will take in the worst case.

9. $rtime_i$: Time in minutes that player i considers job i will take. Previous experience running similar jobs gives player i a better idea of how much time the job will really take, so $min_i \geq rtime_i$.

We explore these results:

1. **Total Value**: The sum of the bids of all selected players, also known as the **Social welfare**, $\mathbf{TV} = \sum_i^n b_i \cdot x_i$.

2. **Total Sum of Payments**: The sum of the calculated payments for all selected players, $\mathbf{TP} = \sum_i^n p_i \cdot x_i$.

3. **Total Weight**: The total weight included. An optimal allocation would result in it being equal to the maximum weight W of the HPC, $\mathbf{TW} = \sum_i^n w_i \cdot x_i$.

4. **Total Time Spent**: Time spent running the experiment measure in seconds, **TTS**.

5. **Average Difference**: The average difference between the bid and the payment for each selected player. We will explore and ponder about these differences in Section 6.3. $\mathbf{AD} = \sum_i^n d_i \cdot x_i$

6. **Pay More Count**: Number of players that would pay more than their bid, $\mathbf{PM} = |\{i : x_i \geq 1 \wedge p_i > b_i\}|$

7. **Same Payment Count**: Number of players that would pay the same amount as their bid, $\mathbf{SP} = |\{i : x_i \geq 1 \wedge p_i = b_i\}|$

8. **Jobs Picked Count**: Number of jobs included, $\mathbf{JP} = \sum_i^n x_i$.

9. **Negative Payment Count**: Number of players that get paid by the mechanism, $\mathbf{NC} = |\{i : x_i \geq 1 \wedge p_i < 0\}|$

In **Experiment 1A** we take $b_i = w_i \cdot min_i$. This choice symbolizes a situation where the player does not know exactly how long her experiment will run for, but she knows the number of required nodes and an upper bound on the maximum running time of the job. Then we evaluate the results using a solver that finds the optimal **Total Value**.

In **Experiment 1B** we follow the same formula for b_i used in **Experiment 1A**; however, instead of using an optimal solver, we use the *Knapsack Greedy Algorithm* described in Section 5.3.1. From now on, we will also call this model, our *ideal Knapsack Auction Model*.

In **Experiment 2B** we take $b_i = (w_i \cdot rtime_i) + (w_i \cdot rtime_i) \cdot 0.1$ to represent an *experienced* player that knows with some 0.1 positive margin of error the

running time of her job. Then, the greedy heuristic uses $\frac{b_i}{w_i \cdot rtime_i}$. We find the results using the greedy solver. We use w_i to test for the maximum capacity of the HPC W.

In **Experiment 3B** we take $b_i = w_i \cdot min_i$ to represent an *inexperienced* player. Then, the greedy heuristic uses $\frac{b_i}{w_i \cdot rtime_i}$. We use the number of nodes requested by the jobs w_i to test for the maximum capacity of the HPC W.

In **Experiment 4B**, we take $b_i = (w_i \cdot rtime_i) + (w_i \cdot rtime_i) \cdot 0.1$ to represent an *experienced* player that knows with some 0.1 positive margin of error the running time of his job. Then, the greedy heuristic uses $\frac{b_i}{w_i}$, as originally defined. We use the number of nodes requested by the jobs w_i to test for the maximum capacity of the HPC W.

We notices during initial testing that some samples defined in Section 6.1 took excessive time to produce a result under the optimal solver, which can be explained by the distribution of our datasource and our choice of b_i. We dropped these samples to simplify experimentation, bu we still include the plots of their distributions them in Appendix C as well as in the project repository. This fact points to another interesting path of experimentation about the distribution of the jobs that arrive at a given time to the scheduler.

We perform a **first round** of experimentation described in Section 6.3.1 with the definitions for experiments previously defined. Then, we proceed to perform a **second round** of experimentation described in Section 6.3.2 executing the same set of experiments but using a modified version of the **Knapsack Greedy Algorithm**. We exclude **Experiment 1A** from this second round, as it does not pertain to the usage of the greedy algorithm or its modified version. Next, we perform a **third round** of experimentation described in Section 6.3.3 where we discard the greedy algorithm and instead base our allocation in a pseudo random value for **Priority** set for each job in the sample. Finally, using the same samples explored in previous rounds of experimentation, we perform a fourth round of experimentation where we run the **Slurm Simulator** [1] to get the results for **Total Value**. We compare the **Total Value** achieved using the simulator with that of the most relevant experiments of previous rounds. These results are described in Section 6.3.4.

6.3 Results

6.3.1 Main Experimentation

Compared to **1A** and **1B**, experiment **4B** achieves a lower **Total Value** because it is using lower *experienced* bids. We can see these differences more clearly in Figure 18.

We see that the **Total Value** achieved in experiments **1A** and **1B** is very similar, as previously predicted by our theoretic foundation. Experiments **1A**, **1B**, and **3B** use similar bids, but for **3B** we altered the greedy algorithm to consider also $rtime_i$ in the denominator to test its effect on welfare maximization. As we can see, Total Value is clearly degraded for **3B**.

Still in the context of **1A**, **1B**, and **3B**; we see in Table 11 that **1A** achieves

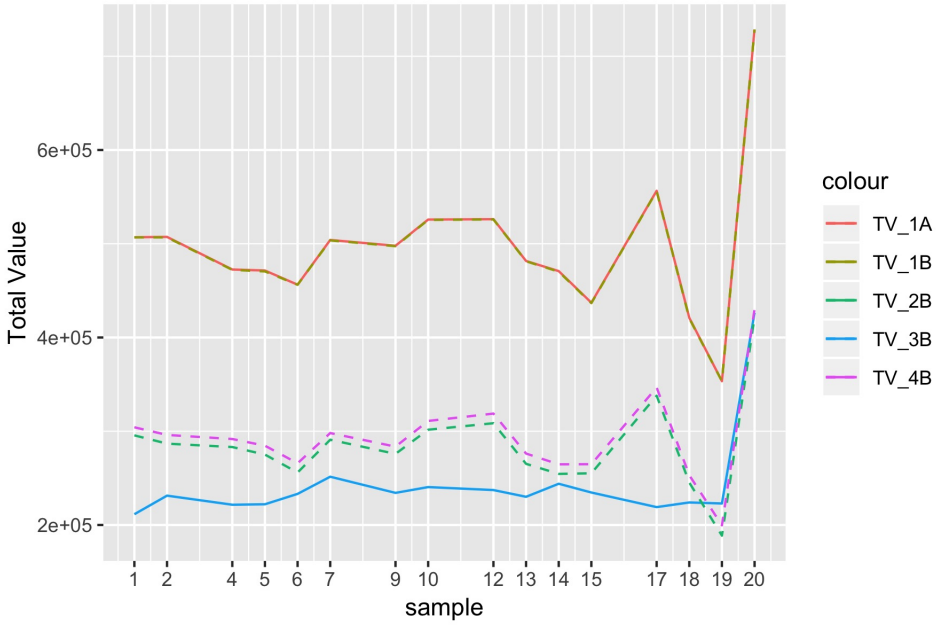

Figure 18: Total Value (Social Welfare) for sample and experiment.

the greater **Total Payment** for almost all samples, with a very large exception for *sample_9*. We see these variations better in Figure 19. Then, we look at Table 12 where we find that the experiments **3B** and **2B** are making players pay more than their bid (or value), thus breaking the requirement that the player should not get negative utility in order for this mechanism to be *DSIC*. It is in *sample_9* where 194 players are forced to pay more than their value, which explains the high value in Table 11.

Moreover, players pay more than what they originally bid because the damage they inflicted on society by participating in the mechanism is larger than what they declared as their value. We see in Table 12 that for experiments **1A**, **1B**, and **4B** the VCG payment rule works as intended and we have that no player *regrets* participating in this auction. Experiments **2B** and **3B** both consider $rtime_i$ in the denominator, we get that some players end up paying more than their bid, this is a clear sign that the formula for the greedy algorithm used for **2B** and **3B** does not produce a *DSIC* mechanism and, subsequently, these settings can be discarded. Until this point, it is reasonable to state that experiments **1A**, **1B**, and **4B** are the ones that more closely represent the model described in Section 5.3.1.

In the context of experiments **2B** and **4B**, we use the same *experienced* bid b_i but differ in the denominator, **2B** considers $w_i \cdot rtime_i$ in the denominator while **4B** only considers w_i. As a result the **Total Value** for these samples is lower for **2B** compared to **4B**. This coincides with what the greedy algorithm

	sample	TV_1A	TV_1B	TV_3B	TV_2B	TV_4B
1	sample_1.txt	**506930**	506850	211480	295615	**304239**
2	sample_2.txt	**507305**	506785	231310	286779	**296064**
3	sample_4.txt	**472530**	472170	221610	283158	**291715**
4	sample_5.txt	**471440**	470480	222105	275120	**284591**
5	sample_6.txt	**456315**	456275	233180	255977	**265832**
6	sample_7.txt	**504035**	503595	251495	290940	**298085**
7	sample_9.txt	**497850**	497410	234255	275866	**283595**
8	sample_10.txt	**525810**	525450	240445	301627	**310999**
9	sample_12.txt	**526210**	525930	237285	308551	**318843**
10	sample_13.txt	**481620**	481300	230075	265248	**276205**
11	sample_14.txt	**470820**	470220	243990	254506	**264714**
12	sample_15.txt	**437110**	436670	234655	255141	**264850**
13	sample_17.txt	**556480**	556120	219080	337897	**346716**
14	sample_18.txt	**421015**	420415	224040	245285	**253124**
15	sample_19.txt	**353420**	353340	222940	188647	**199668**
16	sample_20.txt	**728540**	728540	426377	421757	**430275**

Table 10: Total Value per sample and experiment.

is supposed to do, which is to maximize *Social Welfare*; while the approach of **2B** deviates theoretically from this result by altering the formula of the greedy algorithm. We see a similar trend in Table 11 where the payments calculated for **4B** are greater that those of **2B** in almost all cases, although the difference is not great. Nevertheless, as previously mentioned, **2B** is breaking *DSIC* and even making some players pay more than their original values. Thus, it cannot achieve a better Total Value than **4B**.

From these results, we can say that not only **1A** and **1B** achieve better **Total Value**, but also better behavior guarantees in the form of **Total Payments** and **Number of Players that Pay More** when compared to **3B**. We can affirm the same of **4B** when compared to **2B**.

Although it might have been tempting at first to consider to include $rtime_i$ in the denominator as a way to balance the assignment of resources not only according to the number of nodes but also considering the time they will be used, this inclusion breaks the guarantees of the **Knapsack Greedy Algorithm**.

We notice that the **Total Payment** achieved in Table 11 is the same for experiment **1A** for all the samples except **sample 20**. Samples from 1 to 19 were randomly sampled from our data so they had a high resemblance to the original source. When experimenting with the random samples from 1 to 19, we noticed that all these had the same Total Payment. This surprising result led us to believe that some interesting property of the VCG payment rule could be shown through this study. We found that current literature does not mention the sum of VCG payments, this was a signal that the answer to this mysterious result was in another place, perhaps more experimentation. We looked into the distribution of the picked jobs, in all cases the job with 40 *minutes* was the most

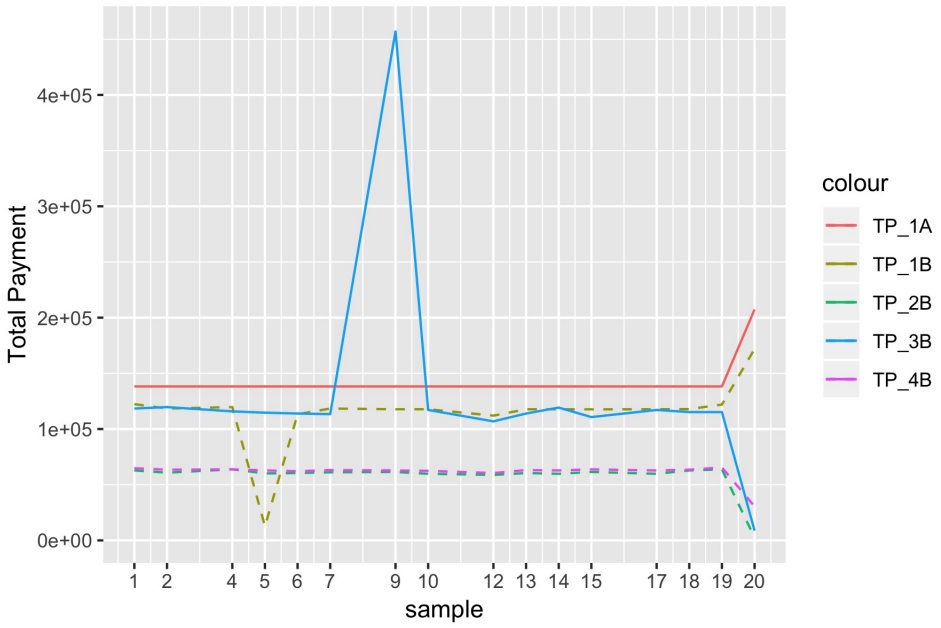

Figure 19: Total Payments per sample and experiment.

	sample	TP_1A	TP_1B	TP_3B	TP_2B	TP_4B
1	sample_1.txt	**138240**	122320	118400	62792	**64796**
2	sample_2.txt	**138240**	118400	119700	60788	**63460**
3	sample_4.txt	**138240**	119680	115840	**63794**	**63794**
4	sample_5.txt	**138240**	12640	114680	60120	**62792**
5	sample_6.txt	**138240**	113280	113920	60454	**62124**
6	sample_7.txt	**138240**	118400	113380	61122	**63126**
7	sample_9.txt	138240	117760	**456960**	61456	**62792**
8	sample_10.txt	**138240**	117760	117120	59786	**62458**
9	sample_12.txt	**138240**	112000	106880	58784	**60454**
10	sample_13.txt	**138240**	117760	113920	60454	**63126**
11	sample_14.txt	**138240**	117760	119240	59786	**62792**
12	sample_15.txt	**138240**	117680	110720	61456	**63794**
13	sample_17.txt	**138240**	117760	117120	59786	**62792**
14	sample_18.txt	**138240**	117880	115200	62792	**63460**
15	sample_19.txt	**138240**	121960	115200	63794	**65464**
16	sample_20.txt	**207360**	171840	8778	2465	**30496**

Table 11: Total Payments per sample and experiment.

	sample	size	PM_1A	PM_1B	PM_3B	PM_2B	PM_4B
1	sample_1.txt	406	0	0	4	5	0
2	sample_2.txt	393	0	0	6	10	0
3	sample_4.txt	415	0	0	4	8	0
4	sample_5.txt	414	0	0	5	9	0
5	sample_6.txt	398	0	0	7	10	0
6	sample_7.txt	396	0	0	5	5	0
7	sample_9.txt	406	0	0	194	6	0
8	sample_10.txt	406	0	0	2	10	0
9	sample_12.txt	425	0	0	2	6	0
10	sample_13.txt	416	0	0	5	8	0
11	sample_14.txt	418	0	0	3	10	0
12	sample_15.txt	430	0	0	2	8	0
13	sample_17.txt	399	0	0	4	9	0
14	sample_18.txt	416	0	0	4	10	0
15	sample_19.txt	403	0	0	7	12	0
16	sample_20.txt	403	0	0	0	0	0

Table 12: Number of players that will pay more than their bid per sample and experiment.

common. We have 3456 nodes, if we multiply by 40 we get the common Total Payment. So, during the calculation of the Total Payment for each sample, the difference was established by those common 40 minutes jobs. To test if this was true, we included a **sample 20** that intentionally excludes 40 *minutes* jobs, the result is that this sample has a different Total Payment value. The reader can review the distributions of these samples in the Appendix C, and compare Figure 45 to any of the other Figures.

Nonetheless, we see that the most common job in **sample 20** has 120 minutes, and if we multiply it by 3456 we do not get **207360**; therefore, the answer to this value might be more complicated than that. We suspect it has more to do with the distribution of the sample. Perhaps those 40 minute jobs were *too* common. Also, notice that min_i is the deciding factor for **1A** and **1B** ordering.

Notice how in Figure 18 the **Total Value** is remarkably higher for *sample_20.txt* and that in Table 11 the **Total Payment** seems to follow a similar trend, except for experiment **4B**. This might indicate that the removal of common *planned* time jobs might have a big influence in the resulting *Social Welfare*. The fact that *sample_20.txt* achieves the higher Total Value for **1A** and **1B** might point out that common *planned* time jobs (40 mins) were hurting the resulting *Social Welfare* of the mechanism. We do not pursue further this path of study, but intuitively, we get the idea that the removal of jobs that are too common can yield an overall benefit.

As a way to measure how accurate our experiments and players bids were, we can look at Table 13 where we see the number of players that end up paying the same as their original bids. We can also consider Table 15, to get an idea

	sample	size	SP_1A	SP_1B	*SP_3B*	*SP_2B*	SP_4B
1	sample_1.txt	406	82	74	121	75	**92**
2	sample_2.txt	393	75	65	111	64	**85**
3	sample_4.txt	415	61	54	117	62	**70**
4	sample_5.txt	414	69	1	118	52	**74**
5	sample_6.txt	398	58	53	101	48	**68**
6	sample_7.txt	396	68	56	109	68	**79**
7	sample_9.txt	406	**72**	65	1	56	67
8	sample_10.txt	406	**65**	58	122	41	61
9	sample_12.txt	425	63	53	108	56	**71**
10	sample_13.txt	416	59	50	105	53	**66**
11	sample_14.txt	418	66	59	122	55	**72**
12	sample_15.txt	430	**83**	68	114	61	82
13	sample_17.txt	399	60	51	110	45	**61**
14	sample_18.txt	416	64	61	108	53	**72**
15	sample_19.txt	403	85	78	103	81	**98**
16	sample_20.txt	403	**22**	13	22	1	2

Table 13: Number of players that will pay their bid per sample and experiment. We can ignore the results for experiment **3B** and **2B**.

of what fraction of players per experiment share this characteristic. However, only by looking at the results per sample, we see that the number of players that end up paying their bid is greater for the experiment **4B** in most cases. Although the number of samples might not give us a clear result, intuitively, this result coincides with the motivation to include experiment **4B**, which is to represent *experienced* players that can tell with high accuracy how long their jobs are going to last.

The results in Table 14 present another way to look at how accurate were the calculated payments. Experiments **1A** and **1B** share the same bid formula, between them we see that the smaller difference is on the side of **1A**, the optimal result. On the other side we have **4B**, which represents lower bids and, thus, reaches lower differences.

We can see how many jobs are included in the Knapsack, or allocated resources for execution, in Table 15. We can ignore experiments **3B** and **2B**. We see that the maximum number of served users per sample is shared between our valid experiments but somewhat leaning towards **1A**, which is the optimal solution that also achieved better **Total Weight** overall as we see in Table 17.

As a sanity check, we have Table 16 that verifies that our VCG payment rule does not end up paying the user, which does not make sense in our context. We have established before that experiments **3B** and **2B** should not be considered because they break an important *DSIC* requirement. In these results, it is curious that these experiments do not also break this rule and subsequently generate negative payments. Although, intuitively, we can infer that the order induced by the modified formula for our invalid experiments still maintained

	sample	AD_1A	AD_1B	*AD_3B*	*AD_2B*	AD_4B
1	sample_1.txt	908.10	**947.12**	234.98	580.66	589.76
2	sample_2.txt	939.10	**988.26**	292.54	593.65	591.87
3	sample_4.txt	805.52	**849.37**	260.46	542.00	549.21
4	sample_5.txt	804.83	**1105.89**	266.49	533.54	535.75
5	sample_6.txt	799.18	**861.80**	309.85	510.73	511.83
6	sample_7.txt	923.72	**972.71**	355.44	592.18	593.33
7	sample_9.txt	885.74	**935.10**	732.45	534.18	543.85
8	sample_10.txt	954.61	**1004.16**	306.61	610.46	612.17
9	sample_12.txt	912.87	**973.95**	309.56	600.50	607.97
10	sample_13.txt	825.43	**873.89**	286.19	509.56	512.21
11	sample_14.txt	795.65	**843.21**	302.61	480.08	483.07
12	sample_15.txt	695.05	**741.84**	290.92	465.02	467.57
13	sample_17.txt	1048.22	**1098.65**	261.35	711.28	711.59
14	sample_18.txt	679.75	**727.25**	267.21	452.65	455.92
15	sample_19.txt	533.95	**574.14**	277.42	329.83	333.01
16	sample_20.txt	1293.25	**1381.39**	1036.23	1040.43	992.01

Table 14: Average difference between bid and payment per sample and experiment. Lower is better.

	sample	size	JP_1A	JP_1B	*JP_3B*	*JP_2B*	JP_4B
1	sample_1.txt	406	216	**217**	255	203	211
2	sample_2.txt	393	211	211	246	213	**212**
3	sample_4.txt	415	**209**	208	249	213	**209**
4	sample_5.txt	414	**218**	**218**	261	211	215
5	sample_6.txt	398	219	**220**	231	207	217
6	sample_7.txt	396	**210**	208	252	216	**210**
7	sample_9.txt	406	**215**	**215**	248	210	212
8	sample_10.txt	406	**204**	203	262	198	201
9	sample_12.txt	425	**224**	**224**	263	214	219
10	sample_13.txt	416	**208**	206	240	206	206
11	sample_14.txt	418	**226**	**226**	259	226	223
12	sample_15.txt	430	**228**	225	253	219	219
13	sample_17.txt	399	**209**	208	234	207	206
14	sample_18.txt	416	224	**225**	242	216	221
15	sample_19.txt	403	**223**	222	237	215	**223**
16	sample_20.txt	403	213	**214**	245	212	211

Table 15: Number of jobs picked per sample and experiment

	sample	size	NC_1A	NC_1B	NC_3B	NC_2B	NC_4B
1	sample_1.txt	406	0	0	0	0	0
2	sample_2.txt	393	0	0	0	0	0
3	sample_4.txt	415	0	0	0	0	0
4	sample_5.txt	414	0	0	0	0	0
5	sample_6.txt	398	0	0	0	0	0
6	sample_7.txt	396	0	0	0	0	0
7	sample_9.txt	406	0	0	0	0	0
8	sample_10.txt	406	0	0	0	0	0
9	sample_12.txt	425	0	0	0	0	0
10	sample_13.txt	416	0	0	0	0	0
11	sample_14.txt	418	0	0	0	0	0
12	sample_15.txt	430	0	0	0	0	0
13	sample_17.txt	399	0	0	0	0	0
14	sample_18.txt	416	0	0	0	0	0
15	sample_19.txt	403	0	0	0	0	0
16	sample_20.txt	403	0	0	0	0	0

Table 16: Number of players that get paid (negative payment) per sample and experiment

some sense of *truthful behavior*. In other words, no job in this ordering became an **altruistic** job, meaning, one with a bid much higher than necessary that raised the value considerably by itself.

We can also see the number of nodes that were assigned for each sample and experiment, also known as the **Total Weight**, in Table 17. The optimal result **1A** achieves total usage, while the others tend to come close or even achieve total usage in some cases. This result is expected because we consider that the greedy algorithm halts when it finds a job that does not fit in the Knapsack, instead of continuing to look for a job that could fit the remaining capacity. In Section 6.3.2, we experiment with this modification.

Another important point about these results is that the algorithm that finds the VCG payments for each included job must also consider that it cannot surpass the amount of covered nodes that the original solution found; otherwise, the outcome will include negative utilities.

So far we have seen that **1A**, which uses the optimal solver, gives overall better results than **1B** that uses the greedy approximation. In Table 18, we see the running times that each experiment has required per sample. Here we can see that some instances require excessively long running times to arrive to a solution. We have seen in Section 2.3 that the scheduler runs in iterations that happen every 60 seconds. If we consider that it is possible that some instances might run for over 60 seconds or around 367 seconds, it is not feasible to pretend an exact optimal solution to the problem. Samples 0,3,8,11, and 16 were excluded from the experimentation space because the solver was not able to find a solution in reasonable time, less than 5 minutes.

	sample	TW_1A	TW_1B	*TW_3B*	*TW_2B*	TW_4B
1	sample_1.txt	3456	3454	3452	3441	3450
2	sample_2.txt	3456	3443	3446	3445	3441
3	sample_4.txt	3456	3447	3447	3447	3450
4	sample_5.txt	3456	3432	3449	3456	3455
5	sample_6.txt	3456	3455	3452	3443	3442
6	sample_7.txt	3456	3445	3456	3456	3445
7	sample_9.txt	3456	3445	3454	3444	3451
8	sample_10.txt	3456	3447	3444	3445	3441
9	sample_12.txt	3456	3449	3448	3441	3449
10	sample_13.txt	3456	3448	3442	3455	3456
11	sample_14.txt	3456	3441	3453	3441	3448
12	sample_15.txt	3456	3445	3442	3442	3447
13	sample_17.txt	3456	3447	3452	3446	3449
14	sample_18.txt	3456	3441	3443	3456	3444
15	sample_19.txt	3456	3454	3446	3441	3455
16	sample_20.txt	3456	3456	3426	3425	3448

Table 17: Total Weight per sample and experiment

	sample	TTS_1A	TTS_1B	TTS_3B	TTS_2B	TTS_4B
1	sample_1.txt	0.91	0.80	0.91	0.70	0.75
2	sample_2.txt	17.05	0.69	0.81	0.70	0.71
3	sample_4.txt	1.76	0.73	0.90	0.77	0.80
4	sample_5.txt	10.86	0.83	0.97	0.73	0.84
5	sample_6.txt	5.62	0.83	0.90	0.74	0.78
6	sample_7.txt	1.71	0.89	1.07	0.82	0.79
7	sample_9.txt	0.88	0.91	0.96	0.74	0.82
8	sample_10.txt	0.84	0.75	1.01	0.70	0.73
9	sample_12.txt	9.65	1.00	1.10	1.01	0.85
10	sample_13.txt	5.41	0.76	0.86	0.72	0.73
11	sample_14.txt	0.79	0.88	1.01	0.85	0.96
12	sample_15.txt	1.00	0.88	0.94	0.80	0.81
13	sample_17.txt	0.79	0.73	0.80	0.71	0.70
14	sample_18.txt	**65.09**	0.86	0.86	0.75	0.80
15	sample_19.txt	**367.67**	0.84	0.88	0.76	0.81
16	sample_20.txt	0.74	0.86	0.88	0.77	0.73

Table 18: Total Time spent per sample and experiment

	sample	TV_1B	TV_1B1	TV_4B	TV_4B1
1	sample_1.txt	**506910**	506850	304308	**304239**
2	sample_2.txt	**507265**	506785	**296247**	296064
3	sample_4.txt	**472530**	472170	**291784**	291715
4	sample_5.txt	**471415**	470480	**284609**	284591
5	sample_6.txt	**456275**	**456275**	**265996**	265832
6	sample_7.txt	**504015**	503595	**298241**	298085
7	sample_9.txt	**497830**	497410	**283663**	283595
8	sample_10.txt	**525750**	525450	**311160**	310999
9	sample_12.txt	**526125**	525930	**318944**	318843
10	sample_13.txt	**481600**	481300	**276205**	**276205**
11	sample_14.txt	**470820**	470220	**264827**	264714
12	sample_15.txt	**437045**	436670	**264982**	264850
13	sample_17.txt	**556480**	556120	**346803**	346716
14	sample_18.txt	**420955**	420415	**253267**	253124
15	sample_19.txt	**353400**	353340	**199686**	199668
16	sample_20.txt	**728540**	**728540**	**430354**	430275

Table 19: Total Value compared between the Knapsack Greedy Algorithm Modified **TV_1B TV_4B** and **TV_1B1 TV_4B1** from the Knapsack Greedy Algorithm.

6.3.2 Knapsack Greedy Algorithm Modified

In these experiments, we modify the original Knapsack Greedy Algorithm defined in Section 5.2.9 so instead of stopping when a job that does not fit is found, it continues until it fills the capacity of the Knapsack or reaches the end of the list of jobs. We compare the results of this setting with the results from the previous round of experiments described in Section 6.3.1.

We define **1B1** and **4B1** as the experiments **1B** and **4B** from Section 6.3.1, respectively.

As we can see in Table 19, the **Total Value** results obtained with the *Knapsack Greedy Algorithm Modified* are better in most cases, but by a small margin. If presented in a plot, we would not be able to distinguish between **1B** and **1B1**, or **4B** and **4B1**. Furthermore, we infer that the differences are due to the fact that the modified version is achieving better usage of the HPC, as shown in Table 20.

It is clear that the modified version achieves a better **Total Value** than the original implementation; however, as wee see in Table 22, there are players that would regret participating in this mechanism because they will end up paying more than their original value for the required resources. This result breaks one requirement for a *DSIC* mechanism, specifically the *Incentive* part.

Finally, the results in Table 23 indicate that in this model the payment rule does not pay the player. Added to the previous results where we see that this modified model achieves better welfare and payments, these results might indicate that the *Knapsack Greedy Algorithm Modified* setting might have some

use. We discuss this possibility in Section 7.

	sample	TW_1B	TW_1B1	TW_4B	TW_4B1
1	sample_1.txt	**3456**	3454	**3456**	3450
2	sample_2.txt	**3456**	3443	**3456**	3441
3	sample_4.txt	**3456**	3447	**3456**	3450
4	sample_5.txt	**3456**	3432	**3456**	3455
5	sample_6.txt	**3455**	3455	**3456**	3442
6	sample_7.txt	**3456**	3445	**3456**	3445
7	sample_9.txt	**3456**	3445	**3456**	3451
8	sample_10.txt	**3456**	3447	**3456**	3441
9	sample_12.txt	**3456**	3449	**3456**	3449
10	sample_13.txt	**3456**	3448	**3456**	**3456**
11	sample_14.txt	**3456**	3441	**3456**	3448
12	sample_15.txt	**3456**	3445	**3456**	3447
13	sample_17.txt	**3456**	3447	**3456**	3449
14	sample_18.txt	**3455**	3441	**3456**	3444
15	sample_19.txt	**3456**	3454	**3456**	3455
16	sample_20.txt	**3456**	**3456**	**3456**	3448

Table 20: Total Weight compared between the modified version **TW_1B TW_4B** and the original version **TW_1B1 TW_4B1**

	sample	TP_1B	TP_1B1	TP_4B	TP_4B1
1	sample_1.txt	**138080**	122320	**71970**	64796
2	sample_2.txt	**137240**	118400	**76399**	63460
3	sample_4.txt	**135300**	119680	**72033**	63794
4	sample_5.txt	**139795**	12640	**68488**	62792
5	sample_6.txt	**134340**	113280	**76356**	62124
6	sample_7.txt	**138905**	118400	**73547**	63126
7	sample_9.txt	**137675**	117760	**70561**	62792
8	sample_10.txt	**140525**	117760	**77461**	62458
9	sample_12.txt	**142705**	112000	**71760**	60454
10	sample_13.txt	**138730**	117760	**70106**	63126
11	sample_14.txt	**137300**	117760	**72010**	62792
12	sample_15.txt	**141560**	117680	**71954**	63794
13	sample_17.txt	**136440**	117760	**72554**	62792
14	sample_18.txt	**136520**	117880	**74619**	63460
15	sample_19.txt	**138780**	121960	**69535**	65464
16	sample_20.txt	**205935**	171840	**37386**	30496

Table 21: Total Payment compared between the modified version **TP_1B TP_4B** and the original version **TP_1B1 TP_4B1**

	sample	PM_1B	PM_1B1	PM_4B	PM_4B1
1	sample_1.txt	2	0	0	0
2	sample_2.txt	3	0	5	0
3	sample_4.txt	0	0	0	0
4	sample_5.txt	71	0	0	0
5	sample_6.txt	0	0	8	0
6	sample_7.txt	4	0	2	0
7	sample_9.txt	2	0	0	0
8	sample_10.txt	6	0	2	0
9	sample_12.txt	10	0	0	0
10	sample_13.txt	0	0	0	0
11	sample_14.txt	0	0	0	0
12	sample_15.txt	19	0	0	0
13	sample_17.txt	0	0	0	0
14	sample_18.txt	0	0	1	0
15	sample_19.txt	3	0	0	0
16	sample_20.txt	0	0	0	0

Table 22: *Number of players that will pay more than their bid* compared between the modified version **PM_1B PM_4B** and the original version **PM_1B1 PM_4B1**

	sample	NC_1B	NC_1B1	NC_4B	NC_4B1
1	sample_1.txt	0	0	0	0
2	sample_2.txt	0	0	0	0
3	sample_4.txt	0	0	0	0
4	sample_5.txt	0	0	0	0
5	sample_6.txt	0	0	0	0
6	sample_7.txt	0	0	0	0
7	sample_9.txt	0	0	0	0
8	sample_10.txt	0	0	0	0
9	sample_12.txt	0	0	0	0
10	sample_13.txt	0	0	0	0
11	sample_14.txt	0	0	0	0
12	sample_15.txt	0	0	0	0
13	sample_17.txt	0	0	0	0
14	sample_18.txt	0	0	0	0
15	sample_19.txt	0	0	0	0
16	sample_20.txt	0	0	0	0

Table 23: *Number of players that will be paid by the mechanism* compared between the modified version **NC_1B NC_4B** and the original version **NC_1B1 NC_4B1**

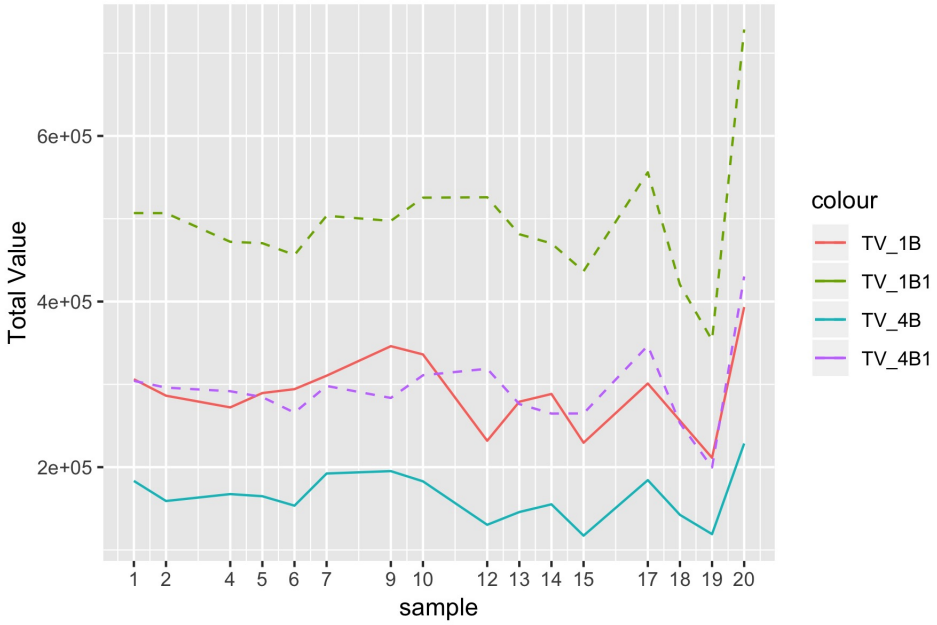

Figure 20: Total Value per sample and experiment using Priority.

6.3.3 Experimentation with Priority

We discard the *Knapsack Greedy Algorithm* completely and instead use a priority list to decide which jobs are considered for resource allocation. This priority list is decided based on the formula: $priority_i = \frac{w_i \cdot 10000}{W} + y$, where y is a uniform random variable between 1 and 10000. We apply this formula to mimic in some way what the *Slurm* calculation of **Priority** would do.

We define **1B1** and **4B1** as the experiments **1B** and **4B** respectively from Section 6.3.1.

	sample	TV_1B	TV_1B1	TV_4B	TV_4B1
1	sample_1.txt	306163	506850	183387	304239
2	sample_2.txt	286326	506785	159129	296064
3	sample_4.txt	272184	472170	167480	291715
4	sample_5.txt	289499	470480	164901	284591
5	sample_6.txt	294235	456275	153504	265832
6	sample_7.txt	310451	503595	192329	298085
7	sample_9.txt	346111	497410	195219	283595
8	sample_10.txt	336130	525450	182885	310999
9	sample_12.txt	231890	525930	130394	318843
10	sample_13.txt	279032	481300	145813	276205
11	sample_14.txt	288413	470220	155136	264714
12	sample_15.txt	229518	436670	117307	264850
13	sample_17.txt	301042	556120	184343	346716
14	sample_18.txt	256163	420415	142429	253124
15	sample_19.txt	211273	353340	119114	199668
16	sample_20.txt	393348	728540	228547	430275

Table 24: Total Value compared between the priority model **TV_1B TV_4B** and the Knapsack Greedy Algorithm **TV_1B1 TV_4B1**

By looking at Figure 20, we get that using this pseudo random Priority value as the main criteria to decide the order of execution or allocation rule, results in lower **Total Value** in both cases, for *inexperienced* bids **1B** and *experienced* bids **4B**. The **Total Payment** values from 25 are also not promising, some of them being negative. Furthermore, there are many players that pay more than their bid, as seen in Table 26, and also many instances where players will be paid by the mechanism (Table 27), suggesting that they are regarded as **altruistic** by this odd mechanism.

	sample	TP_1B	TP_1B1	TP_4B	TP_4B1
1	sample_1.txt	116060	122320	61535	64796
2	sample_2.txt	-129740	118400	-124659	63460
3	sample_4.txt	4099400	119680	3930069	63794
4	sample_5.txt	776786	12640	132268	62792
5	sample_6.txt	210874	113280	117567	62124
6	sample_7.txt	338780	118400	218388	63126
7	sample_9.txt	177904	117760	109746	62792
8	sample_10.txt	192587	117760	84541	62458
9	sample_12.txt	-194261	112000	-146102	60454
10	sample_13.txt	188096	117760	112801	63126
11	sample_14.txt	384991	117760	233289	62792
12	sample_15.txt	210931	117680	106892	63794
13	sample_17.txt	1477868	117760	859302	62792
14	sample_18.txt	383670	117880	163162	63460
15	sample_19.txt	433860	121960	143637	65464
16	sample_20.txt	493232	171840	315593	30496

Table 25: Total Payments compared between the priority model **TP_1B TP_4B** and the Knapsack Greedy Algorithm **TP_1B1 TP_4B1**

	sample	PM_1B	PM_1B1	PM_4B	PM_4B1
1	sample_1.txt	44	0	60	0
2	sample_2.txt	5	0	27	0
3	sample_4.txt	161	0	163	0
4	sample_5.txt	162	0	148	0
5	sample_6.txt	43	0	70	0
6	sample_7.txt	122	0	139	0
7	sample_9.txt	45	0	65	0
8	sample_10.txt	86	0	113	0
9	sample_12.txt	3	0	6	0
10	sample_13.txt	54	0	71	0
11	sample_14.txt	176	0	161	0
12	sample_15.txt	78	0	92	0
13	sample_17.txt	53	0	71	0
14	sample_18.txt	149	0	164	0
15	sample_19.txt	157	0	75	0
16	sample_20.txt	170	0	158	0

Table 26: Players Paying More than their bids compared between the priority model **PM_1B** and the Knapsack Greedy Algorithm **PM_4B** and **PM_1B1 PM_4B1**

	sample	NC_1B	NC_1B1	NC_4B	NC_4B1
1	sample_1.txt	0	0	7	0
2	sample_2.txt	136	0	129	0
3	sample_4.txt	44	0	46	0
4	sample_5.txt	14	0	14	0
5	sample_6.txt	17	0	0	0
6	sample_7.txt	0	0	0	0
7	sample_9.txt	0	0	0	0
8	sample_10.txt	17	0	17	0
9	sample_12.txt	129	0	129	0
10	sample_13.txt	0	0	0	0
11	sample_14.txt	15	0	15	0
12	sample_15.txt	0	0	0	0
13	sample_17.txt	6	0	1	0
14	sample_18.txt	3	0	0	0
15	sample_19.txt	0	0	3	0
16	sample_20.txt	3	0	3	0

Table 27: Players payed by the mechanism compared between the priority model **NC_1B NC_4B** with the main results that use the Knapsack Greedy Algorithm **NC_1B1 NC_4B1**

6.3.4 Experimentation with Slurm Simulator

We use the **Slurm Simulator** to run our samples and produce what would be a real output from the *Slurm* Scheduler. Then, we use the formula $v_i = w_i \cdot min_i$ to calculate the value for each job i and get a **Total Value** achieved by the resource allocation outcome of the simulator. Then, we compare it to the experiments **1B** from Section 6.3.1 (Main Experimentation) that we call **1B1**, **1B** from Section 6.3.2 (Knapsack Greedy Modified) that we call **1B2**, and **1B** from Section 6.3.3 (Priority) that we call **1B3**.

The *Slurm* Simulator computes a total simulation for the whole sample, meaning that jobs that could not be allocated in a first iteration, are then scheduled after other jobs finish, but with a higher **start time**. To establish a common ground between the Simulator and our models, we iterate through the result of the Simulator ordered by **start time** ascending. In this way, we try to get the scheduling result for the first iteration of the Simulator.

As we can see in Figure 21, the Simulator achieves results closer to the **Priority** model than the other models. This might indicate that the intuition behind our pseudo random Priority calculation was somewhat closer to the original calculation.

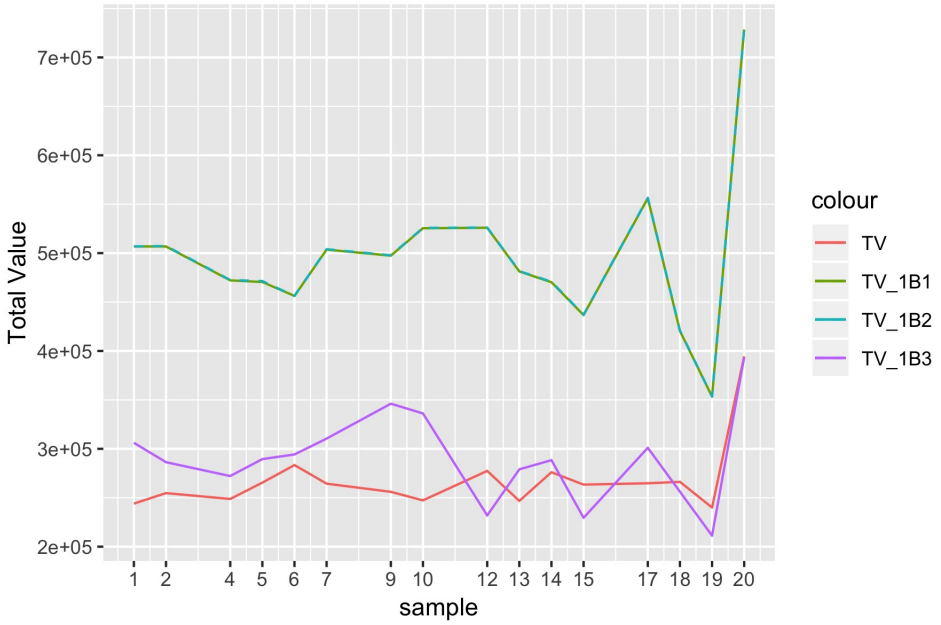

Figure 21: Total Value per sample and experiment from the *Slurm* Simulator compared to previous results.

	sample	TV	TV_1B1	TV_1B2	TV_1B3
1	sample_1.txt	244007	506850	506910	306163
2	sample_2.txt	254720	506785	507265	286326
3	sample_4.txt	248748	472170	472530	272184
4	sample_5.txt	265446	470480	471415	289499
5	sample_6.txt	283402	456275	456275	294235
6	sample_7.txt	264376	503595	504015	310451
7	sample_9.txt	256051	497410	497830	346111
8	sample_10.txt	247317	525450	525750	336130
9	sample_12.txt	277427	525930	526125	231890
10	sample_13.txt	246755	481300	481600	279032
11	sample_14.txt	276072	470220	470820	288413
12	sample_15.txt	263423	436670	437045	229518
13	sample_17.txt	264798	556120	556480	301042
14	sample_18.txt	266251	420415	420955	256163
15	sample_19.txt	240015	353340	353400	211273
16	sample_20.txt	394505	728540	728540	393348

Table 28: Comparison between the Total Value results achieved by the *Slurm* Simulator and previous models **TV_1B1 TV_1B2 TV_1B3**

	sample	TV	TV_1B	PoA
1	sample_1.txt	244007	506850	2.08
2	sample_2.txt	254720	506785	1.99
3	sample_4.txt	248748	472170	1.90
4	sample_5.txt	265446	470480	1.77
5	sample_6.txt	283402	456275	1.61
6	sample_7.txt	264376	503595	1.90
7	sample_9.txt	256051	497410	1.94
8	sample_10.txt	247317	525450	2.12
9	sample_12.txt	277427	525930	1.90
10	sample_13.txt	246755	481300	1.95
11	sample_14.txt	276072	470220	1.70
12	sample_15.txt	263423	436670	1.66
13	sample_17.txt	264798	556120	2.10
14	sample_18.txt	266251	420415	1.58
15	sample_19.txt	240015	353340	1.47
16	sample_20.txt	394505	728540	1.85

Table 29: Price of Anarchy based on Total Value of our samples.

6.4 Comments on the Price of Anarchy

Through experimentation we have seen that it is possible to achieve a near optimal **Total Value** for our model in a computationally efficient way. We have also run our samples in a *Slurm* Simulator to get results as close as possible to what the real *Slurm* might have returned. Then, we calculated the Total Value for the results of the simulator based on the same ideas applied for our near optimal model. In this way we have established a **common ground**, the Total Value, that we can use to compare the outcomes of our models to the outcome of the simulator.

Having said that, we can now take a look at the results in Table 28 and make a comparison between the results of **TV**, the simulator, and **TV_1B1**, our **ideal** *Knapsack Auction Model* defined in Section 5.3.1. Considering this comparison and **Total Value** as a common value that we can calculate, we are now able get a reasonable approximation of the **Price of Anarchy**.

If we average the **PoA** results presented in Table 29, we get that $PoA \approx 1.85$. Then, we can say that an ideal outcome of the *Slurm* scheduling game might be around 1.8 times better than a *usual* outcome.

Notice that the Price of Anarchy is defined considering the worst equilibrium, but we are only considering a non optimal (*usual*) equilibrium as a worst equilibrium. We make this simplification because it is difficult to imagine a situation or job input distribution that would make the scheduler result in a worse outcome than it usually returns. Nevertheless, to consider the influence of different distributions of inputs is an interesting perspective to work on in the future.

6.5 Comments on the Viability of Cooperation

The *ideal Knapsack Auction Model* does not allow for the possibility of cooperation in the sense that users could reach some agreement and play a strategy that will increase their utilities as a coalition. In our model, the best strategy of users is to selfishly send their jobs stating their true valuations as bids; however, is not this also a form of cooperation?

Remember the **ISP Routing Game** defined at the beginning of Section 5.1. In this model the users had a dominant strategy: to act selfishly in order to get some utility or decrease their costs. What if their dominant strategy was to inevitably cooperate to get an even greater utility or a lower cost?

The *Knapsack Auction Model* is an ideal game or auction. The dominant strategy for the players under this model is to tell the truth, and thus, achieve their optimal utility; not only that, but the whole mechanism achieves optimal *Social Welfare*.

We argue that telling the truth, or being forced to tell the truth, is also a form of cooperation. It is our *Knapsack Auction Model*, our **design choice**, that imposes the necessary constraints on users, so we achieve the desired guarantees of: *DSIC*, maximum *Social Welfare*, and *Computationally Efficient* calculation of the outcome. In short, the mechanism is making players cooperate for the greater good because that is the only choice players have in this scenario. That is why we call it an **ideal** auction or model.

It is not trivial how to translate the desired guarantees achieved by the *Knapsack Auction Model* to the *Slurm* Scheduler. The results of this study hint that the *planned* running time that the player establishes for her job plays a key role in the calculation of execution ordering.

Looking back at our data source, we notice that there is usually a significant difference between the *planned* running time and the real *running* time. For example, in the case of the 40 minutes jobs from our data source, which are the most common jobs in all our samples, except *sample_20*. We see in Figure 22 that most of these jobs have a *real* running time half of the *planned* running time, and that they usually require around 16 nodes. We have also seen that among the **Total Payment** results for experiment **4B** that uses *experienced* bids (Table 11), it is *sample_20* the one that achieves the best results for the players, meaning lower payments. Not only that, but we can see from our results, comparing Table 13 and Table 15, that for this sample almost all participants payed less than their values.

Looking back at the formula of the *Knapsack Auction Model*, if we simplify it, we see that the dominant factor is min_i, or $rtime_i$ for the *experienced* case. This coincide with the way the **backfill** procedure works, and how we can intuitively say it can reach better allocations. Backfill considers the *planned* running time to calculate if a job can be allocated resources even if its Priority is not high enough, with the purpose of filling some idle resources. If the player sends *planned* running time closer to *real* running time, then, we can infer that the backfill procedure will have more chances to allocate these jobs in *tight* time spaces.

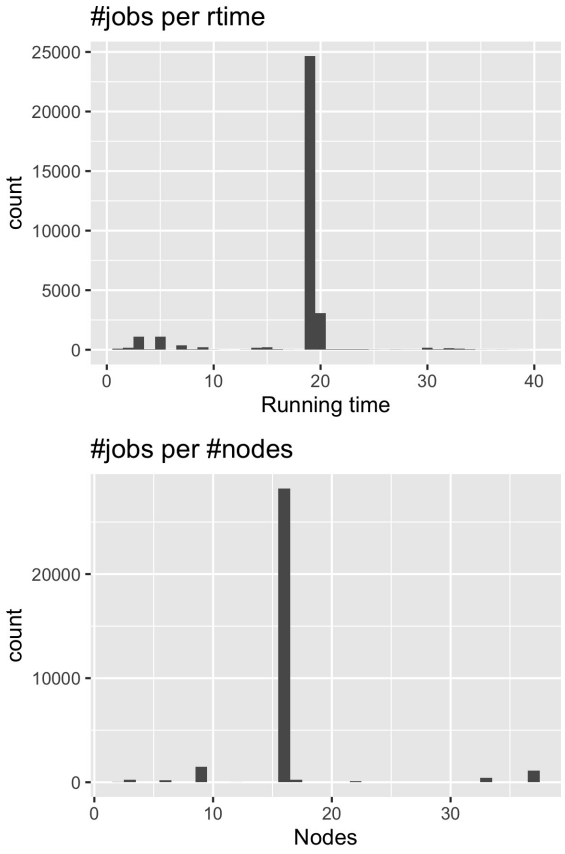

Figure 22: Distributions from source for 40 minutes (wallclock) *planned* running time jobs. Top: Number of jobs per rtime. Bottom: Number of jobs per number of nodes requested.

In conclusion, we can say that *more realistic* or *carefully considered planned* running times are a way in which we can translate the desired guarantees achieved by the *Knapsack Auction Model* to the *Slurm* Scheduler. Furthermore, we posit that by *carefully adjusting* the *planned* running times for their jobs, players (users) would be cooperation with the increase in *Social Welfare* of the system.

7 Conclusions

We developed and presented a detailed study of the main factors that play in the scheduling process of *Slurm*. We describe how these factors interact with each other through the use of practical examples and the description of the main algorithms used by the scheduler. We describe the options available in *Slurm* for the configuration of the scheduler. We consider that these tasks were fundamental in the understanding of the system we wanted to study because the current documentation of *Slurm* regarding the scheduler for the version **17.11.7** can be disperse and it might be difficult for some readers to clearly grasp the main ideas behind the *Slurm* Scheduler. The developers of *Slurm* present a wide array of documentation sources, including presentations of various topics related to Slurm, or even specific to the scheduling process. We attempt to centralize this information in our description of the mechanism. The result of this analysis is described in Section 2.

We present a summary (Section 5.1) of some of the main ideas we used for our project from the field of **Mechanism Design**, also including relevant background theory from **Algorithmic Game Theory**. The book *Twenty Lectures on Algorithmic Game Theory*[10] presents *Mechanism Design* with an economic perspective and relevant case studies, which are helpful in the understanding of its concepts; however, it omits an introduction to basic concepts of *Game Theory*, the author acknowledges this fact in the *Introduction* of the book. The chapter on *Algorithmic Mechanism Design* from the book *Algorithmic Game Theory*[9] starts with a *Voting Systems* perspective and then develops into a more economic perspective, it is a good extension to the ideas presented in *Twenty Lectures on Algorithmic Game Theory*. Moreover, the book *Algorithmic Game Theory* presents a great introduction to the basic concepts of *Game Theory*. We have selected the theory for our summary and presented it in an order that should allow a reader, without a strong background on the field, to grasp the main ideas used for this project.

We have shown through experimentation that the payment rule defined by a *VCG mechanism* renders the allocation rule defined by the *Knapsack Greedy Algorithm* implementable. Thus, making our model *DSIC*. Moreover, we have shown that the allocation rule maximizes *Social Welfare*, which is a desired objective. Furthermore, the allocation rule is *Computationally Efficient*, and the results are close to the optimal. We also calculate these optimal results using a specific solver.

We have shown that our *ideal* auction, the *Knapsack Auction Model*, achieves

the best results regarding resource allocation in reasonable time, among other desired guarantees. Also, from the variants of our model, the *Knapsack Greedy Algorithm Modified* is the one that best resembles the *backfill* procedure that attempts to allocate as many available resources as possible. We have seen that the payments calculated under this model result in some players having to pay more than their value, which breaks the *DSIC* condition. However, this increase in payment can be seen as a sort of *real* metric of the damage players inflict on society by participating; because their participation is preventing other users to be potentially *backfilled*.

Through the application of *Algorithmic Game Theory* and specifically *Mechanism Design* tools, we have designed a model that captures some of the most important characteristics of the *Slurm* Scheduler. By testing this model and its variants, we have shown how some variables established by the users of the system have influenced the outcome of the mechanism. From these results, we have argued that there is a way to translate the desired guarantees achieved by our model into the *Slurm* mechanism. In summary, *carefully considering planned* running times might result in an increase in *Social Welfare* for all users of *Slurm*. We consider this as a form of cooperation, or forced cooperation, where the design choices of the mechanism impose this behavior on users. From this idea, we expand our thoughts on the **Viability of Cooperation** in Section 6.5.

The **Autosubmit** library manages workflows in the HPC platform *Marenostrum4*. We developed a data collection tool that retrieves important experiment and job information from the users of this library. We analyzed the data obtained using this tool and were able to generate a reliable data source that was later used for sampling and experimentation. The process of cleaning the data obtained using the tool is thoroughly documented in Section 4.

We have shown through experimentation that the *Social Welfare* achieved by our *ideal* auction, the *Knapsack Auction Model*, is greater than that of any of the other proposed variants of this model. Then, we have seen that an allocation rule based on **Priority**, which attempts to allocate resource in a *fair* way, achieves a lower *Social Welfare* than that of a mechanism that implements an allocation rule that gives resources to those who value them the most. We use this fact to argue about an *approximate* value of the **Price of Anarchy** for the *Slurm* Scheduler that we present in Section 6.4. We conclude that the *Slurm* Scheduler might be able to allocate resources 1.8 times better than it usually does. A way to increase the performance of resource allocation could be by adjusting the fairness imposed by the factors that influence Priority, by including the idea of a limited *currency* that users spend in exchange for resources to run their jobs. This currency can be compared to the idea of *Usage* that is currently included in the calculation of Priority.

The results achieved with the *ideal Knapsack Auction Model* comply with the guarantee we established in our theory summary, which says that if $m_i \leq \beta \cdot W$ for every player i and $\beta \in (0, \frac{1}{2}]$, then the approximation guarantee increases to $1 - \beta$. From our samples we have that $\beta \approx 0.0174$ and our approximation guarantee is around $1 - \beta \approx 0.9826$. Comparing our results for **1A** and **1B** in

Table 10, we see that this approximation guarantee boundary is satisfied.

We have seen that our *special sample*, where we do not consider 40 minutes jobs, achieves very different results than the rest of samples. This suggests that the distribution of the input might play an important role in the scheduling process.

8 Future Work

Experimentation has hinted that the distribution of the jobs that arrive at a certain time to the scheduler plays an important role in the result of the resource allocation process. A next step is to pursue more analysis of these distributions to find if some of them favor certain outcomes.

One of the main purposes of this study was to find if there exists some way of *cooperation* that would benefit users more than if they played selfishly. The models developed in the study only applied to selfish actors, but we argued that there is indeed some sense of cooperation in our *ideal* model. Therefore, a next step would be to study a model that takes as input not only users and its jobs, but also a mapping of users to coalitions of users.

The results from our experimentation have hinted that the inclusion of some kind of *currency* in the calculation of **Priority** for the *Slurm* Scheduler might result in better guarantees for achieving a greater Social Welfare. We posit that there are two ways to try this idea: First, to include it as another factor for the calculation of Priority; second, to replace the *Usage* factor by converting it into a budget, and then, let the users pay for the resources they request with currency from that budget. Then, following on the second approach, it would be interesting to develop and test a scheduler based on the *Knapsack Greedy Algorithm Modified* where users spend their assigned budgets. Then, analyze the results and see if real users arrive at the *Dominant Strategy Equilibrium*, and if so, how long it takes them to arrive to that point.

We have looked at results for players that pay more than their bid, or that pay the same, but we have not analyzed the results for players that pay less than their bid. The reason that these players pay less is generally explained by the *VCG payment rule*; however, it would be interesting to analyze how the distribution of the bids of the other players affects this result.

The variants **3B** and **2B** of our approximation model broke the no negative utility requirement for a *DSIC* mechanism. However, these variants still maintained that the players are not paid by the mechanism, and even achieving closer **Total Payments** to those of their *valid* counterparts. Furthermore, **3B** achieved a way bigger **Total Payment** for *sample_9.txt*, but way smaller for *sample_20.txt*. It might be useful to identify how these variants of our model can impose other guarantees.

For our analysis and approximation of the **Price of Anarchy** we used the idea of an *usual* equilibrium because it is not clear how a worst equilibrium is constructed. It can certainly be useful to document under which conditions a worst equilibrium might occur.

References

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

[2] Baraglia R., Capannini G., Pasquali M., Puppin D., Ricci L., Techiouba A.D. "Backfilling Strategies for Scheduling Streams of Jobs On Computational Farms". In: *Making Grids Work* (2008).

[3] Earth Science Department at BSC-CNS. *Autosubmit*. URL: `https://autosubmit.readthedocs.io/en/latest/`.

[4] D. Braess, A. Nagurney, and T. Wakolbinger. "On a paradox of traffic planning". In: *Transportation Science* 39 (2005), pp. 446–450.

[5] S. Gill Williamson Edward A. Bender. *Foundations of Combinatorics with Applications. Chapter 9: Rooted Plane Trees*. Dover, 2006.

[6] *Google OR-Tools*. 2020. URL: `https://developers.google.com/optimization/reference/python/algorithms/pywrapknapsack_solver`.

[7] *MareNostrum4 User's Guide*. 2020. URL: `https://www.bsc.es/user-support/mn4.php`.

[8] John Nash. "Non-Cooperative Games". In: *Annals of Mathematics* 54 (1951), pp. 286–295.

[9] Noam Nisan, Tim Roughgarden, Éva Tardos, Vijay V. Varizani. *Algorithmic Game Theory*. Cambridge University Press, 2007.

[10] Tim Roughgarden. *Twenty Lectures on Algorithmic Game Theory*. Cambridge University Press, 2016.

[11] Sergei Leonenkov, Sergey Zhumatiy. "Introducing new backfill-based scheduler for SLURM resource manager". In: *Procedia Computer Science* 66 (2015), pp. 661–669.

[12] *Slurm Documentation*. URL: `https://slurm.schedmd.com/archive/slurm-17.11.7/overview.html`.

A Slurm Simulator

Citing directly from the web page[1] of the *Slurm* Simulator:

Having a precise and a fast job scheduler model that resembles the real-machine job scheduling software behavior is extremely important in the field of job scheduling. The idea behind Slurm simulator is preserving the original code of the core Slurm functions while allowing for all the advantages of a simulator. Since 2011, Slurm simulator has passed through several iterations of improvements in different research centers. We took last available version's code and we fixed various issues, we improved the simulator's performance of about 2.6 times, made it deterministic across several same set-up runs, and improved the its accuracy.

A.1 Setup Process

It is highly recommended to install the simulator in a fresh Linux installation. This process was performed in an virtual machine running **openSUSE Leap 15.1**.

The simulator's main repository is `https://github.com/BSC-RM/slurm_simulator_tools`, there you can find general installation and execution instructions. However, if you only consider what is mentioned in the description of the repository, you might encounter a couple surprises.

In your Linux virtual machine, as `root`, do the following before attempting to install the simulator[2]:

1. Install mariadb `zypper install mariadb-server` and developer package `zypper install mariadb-devel`. It might be already installed as mysql.
2. Enable mariadb server `systemctl enable mariadb`
3. Start mariadb server `systemctl start mariadb`
4. Install python `zypper install python`
5. Install python3 `zypper install python3`
6. Install python3 developer `zypper install python3-devel`
7. Install the latest `pip` for each python installations.
8. Install python packages `pip install pymysql, pandas`
9. Install R `zypper install R`
10. Install GCC `zypper install gcc`

[1] https://www.bsc.es/research-and-development/software-and-apps/software-list/slurm-simulator

[2] Some steps might be unnecessary

11. You might need to create a *Slurm* user in your database: `create user 'slurm'@'localhost' identified by 'slurm';`

These steps should cover all the system requirements necessary to install the simulator. Lets proceed with the installation process:

1. Clone the repository of the simulator into your home directory: `git clone https://github.com/BSC-RM/slurm_simulator_tools`
2. Inside the cloned folder, run `./install_slurm_sim.sh`. This operation will take a while and you will receive a lot of information feedback from the installation in your terminal. At some point the installation might fail because there is some missing requirement. It will tell you in the terminal which package is missing. Install the package and retry the installation.
3. Once the installation is complete you do not really see a confirmation message, but there should not be an error message either.
4. The installation process has copied `https://github.com/BSC-RM/slurm_simulator` inside the main folder.

Everything should be working fine now, but if you get an error or strange behavior while testing your traces, it is probably because something failed during the installation process. Try to install again and pay attention to any error message you get once the installation is completed.

A.2 Workload Generation

This process can be a little bit tricky. You will have to compile some `C` code.

Navigate to `/slurm_simulator/contribs/simulator`. Inside that folder you will find the file `swf2trace.c`. This is a sort of parser that converts your input from a text file to a `trace` file. Open it to see the details of how it works. It takes as input a text file with format (ordered as position per line):

1. JobID
2. Submit time
3. 0 (not relevant)
4. Running time in minutes (exact running time of job)
5. -1 (not relevant)
6. -1 (not relevant)
7. -1 (not relevant)
8. Total number of cores
9. Planned time in minutes (What the user thinks it will take.)

10. -1 (not relevant)
11. 1 (not relevant)
12. 1 (not relevant)

Each job should be organized considering these values separated by commas (you can define the separator inside `swf2trace.c`); 1 line per job. All values are *Integer*. It is important to leave an empty line at the end of your file; otherwise, the parser will ignore the last job in your list. This happens because of the function used by the parser to read the input file.

There are a couple of necessary fixes that you can apply to the original `swf2trace.c` file. For example, the number of lines to read is a constant *large* value. Also, you can implement your own modifications. Anyway, after you are done modifying it, remember to compile: `gcc swf2trace.c -o swf2trace`.

Now you are ready to generate your trace using your text files and applying the format described before, run `./swf2trace PATH_TO_FILE/yoursample.txt` and you will get a file called `simple.trace` in the same folder as `./swf2trace`. Copy it wherever you want, and make sure to change its name to something more recognizable like `yoursample.trace` and also move it to a place where you could easily retrieve it.

A.3 Testing Process

The hard part was to generate your trace, now comes the easy part.

Go to your main simulation folder, it should be something like `ROOT_PATH/slurm_simulator_tools/`

Then, run `./run_simulation_local.sh PATH_TO_FILE/yoursample.trace` and that is basically it. You will see some feedback on your terminal while your simulation is performed. Sometimes it gets stuck and you will notice that no progress is performed, stop it and launch it again.

After your simulation is finished, you will see a new folder which name starts with the name of your trace file followed by some numbers and ending in `.trace` in your main simulation folder. Inside this result folder, the most important file for us is the one ending in `.csv`, those are our results in a comprehensible *CSV* format.

A.4 Changing Cluster Configuration

To change the configuration of your cluster, including the number of nodes, memory, scheduling policy, etc. Go to:

`ROOT_PATH/slurm_simulator_tools/install/slurm_conf/`

Once there, open the file `slurm.conf.template` and modify it according to your preferences. You can find detailed descriptions of the available options in the *Slurm Administrators* section of the *Slurm* Documentation[3].

[3] `https://slurm.schedmd.com/archive/slurm-17.11.7/`

Save your changes and proceed to run your simulations again. For the purposes of this project we copied the node configuration of *Marenostrum4* and it worked without problems.

B Experimentation Environment

In this section, we give a general description of our experimentation environment. We start by describing the main classes of our program and describing their main methods. For a more detailed description of every variable and method, go to the repository `https://earth.bsc.es/gitlab/wuruchi/tfm_agt` where you can also find all the code involved in data generation and analysis of results, among other resources. Our experimentation environment was developed using **Python3**.

B.1 Structure

B.1.1 Job

This class represents a job in the context of scheduling. It captures the main information that we need from a job, such as the number of nodes, the minutes it is supposed to run, the value it has for its owner, etc. An important constant is `EXPERIENCE` that represents the percentage of extra time a user assigns to the running time of her job to guarantee its completion. We suppose the user calculates this percentage based on her experience running similar jobs.

B.1.2 Platform

This class provides the main methods to parse data and make it available for the `Scheduler` class. It can read the data source and generate samples from it, or it can read a sample and prepare it for scheduling.

Methods:

- `source_size`: Returns the number of jobs in the platform. For example, the number of jobs in a sample.
- `experiment_ingestion`: Reads a sample file and converts it into an internal `Job` list.
- `print_jobs`: Prints a string representation of every job in the `Job` list.
- `print_summary`: Prints a summary of the `Job` list.
- `random_sample`: Generates a random sample from the current `Job` list and saves it in a file.
- `special_sample`: Generates a special sample from the current `Job` list and saves it in a file.
- `add_priority`: Adds the priority value to each job in the `Job` list and saves it in a file.
- `generate_trace`: Uses the current `Job` list and generates a trace text format that can be read by the *Slurm* Simulator.

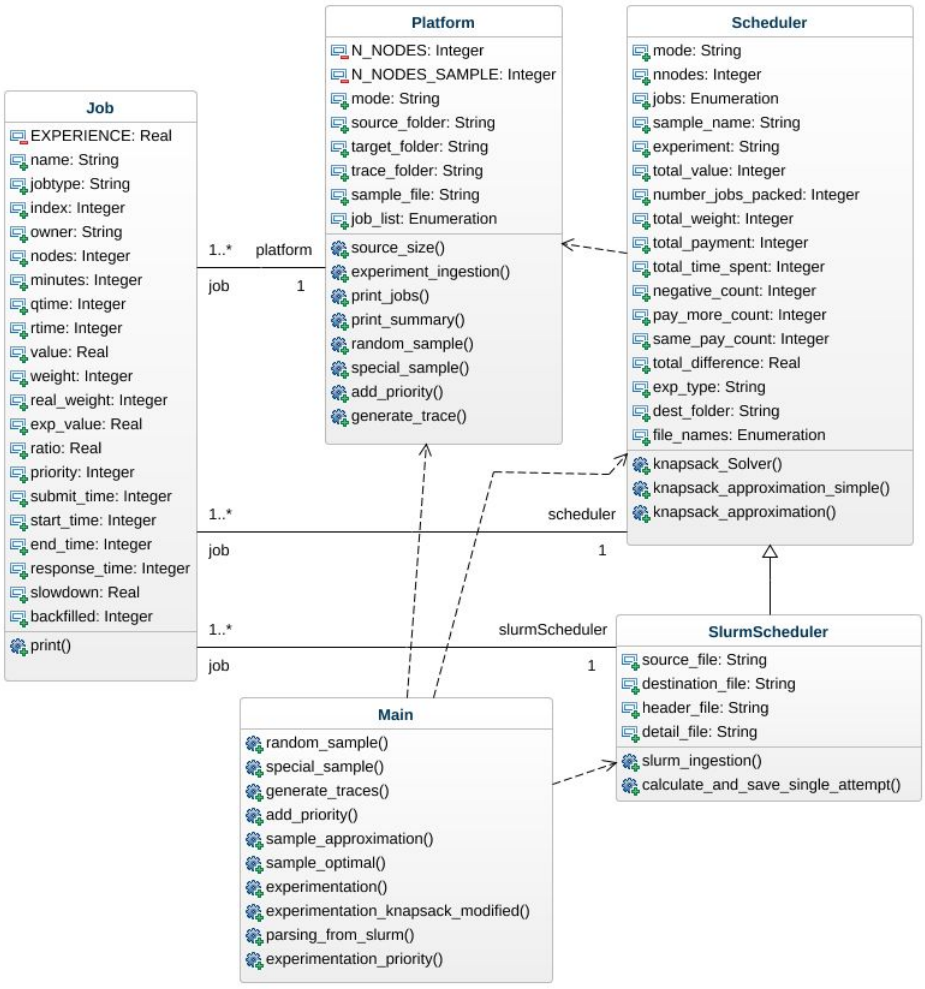

Figure 23: Class diagram of the testing platform.

B.1.3 Scheduler

This class implements in its methods the main scheduling procedures (for our models) that we test in our project. It uses the information provided by `Platform` to produce an scheduling outcome.

Methods:

- `knapsack_Solver`: Solves the knapsack instance using an optimal solver.
- `knapsack_approximation_simple`: A function that helps the calculation performed by `knapsack_approximation`.
- `knapsack_approximation`: Solves the knapsack instance using the Knapsack Greedy Algorithm, also implements the variants we studied.

B.1.4 SlurmScheduler

This class is in charge of handling the data generated by the *Slurm* Simulator. Its main task is to read the result data and calculate the value of the scheduling process in the same terms generated by the class `Scheduler` methods.

Methods:

- `slurm_ingestion`: Reads the `.csv` file generated by the *Slurm* Simulator and saves it in an internal `Job` list.
- `calculate_and_save_single_attempt`: Using the internal `Job` list, it generates a solution comparable to those generated by our models implemented by `Scheduler`.

B.1.5 Main

This class represents our main workflow. The methods of this class represent the main steps we have taken during the experimentation process developed for this project.

Methods:

- `random_sample`: Performs the random sample operation and saves the results as text files in a folder.
- `special_sample`: Generates the special sample and saves it as a text file in a folder.
- `generate_traces`: Reading the samples from text files, generates the traces in the format that the *Slurm* Simulator can read and saves them as text files in a folder.
- `add_priority`: Reading the samples from text files, calculates and adds the priority value. Then, it stores these modified samples as text files in a different folder.

- `sample_approximation`: Reads a sample, processes it, and calculates the result of the scheduling process using a defined variant of the *Knapsack Greedy Algorithm*.
- `sample_optimal`: Reads a sample, processes it, and calculates the result of the scheduling process using the *Optimal Algorithm*.
- `experimentation`: Performs the experimentation process for our first experimentation round. *Main experimentation* in Section 6.3.1.
- `experimentation_knapsack_modified`: Performs the experimentation process for our second experimentation round. *Knapsack Greedy Modified* in Section 6.3.2.
- `experimentation_priority`: Performs the experimentation process for our third experimentation round. *Experimentation with Priority* in Section 6.3.3.
- `parsing_from_slurm`: Performs the experimentation process for our fourth experimentation round. *Experimentation using the Slurm Simulator* in Section 6.3.4.

B.2 Workflow

Our source data file is `jobs_subset_mn4.txt`, where the results from our data source analysis and cleaning is stored. Then, our main workflow can be described as the following sequence:

1. Generates the random samples from the data source by executing the method `random_sample`.
2. Generate the special sample by executing `special_sample`.
3. Generate the samples with priorities using `add_priority`, these results are stored in a separate folder so they do not mix up with the original samples.
4. At this point you can generate the traces by executing `generate_traces`. We copy these traces to our virtual machine where the *Slurm* Simulator has been installed as described in Appendix A. We run the simulation for each trace and then copy the results (the `.csv` file) from the virtual machine to our experimentation environment.
5. Now you are ready to perform the main experimentation round by using the method `experimentation`. **However**, during this process you might notice that some samples take too much time for the optimal solver, because of the *NP* nature of this problem. We recommend to avoid these samples. Inside `experimentation` you can specify which samples to consider. Through experimentation, we have seen that around 4 or 5 samples will not run in under 5 minutes for the optimal solver.

6. You have identified *valid* samples. We continue with our experimentation process by executing `experimentation_knapsack_modified`.

7. Then we proceed with `experimentation_priority`. We have now went through the main 3 rounds of experimentation defined by our models and implemented in our tool.

8. We proceed to execute the final round of *experimentation* that consists on reading the results retrieved from the *Slurm* Simulator and producing a comparable result. For this purpose, we execute `parsing_from_slurm`.

You can find the folder `jobs_analysis` inside the project repository. Then, inside this folder, you can find the file `Experiments_Analysis.Rmd` that includes, apart from the whole data source generation and cleaning process, the tools to evaluate the results, generate plots, tables, and some other relevant functions.

C Samples

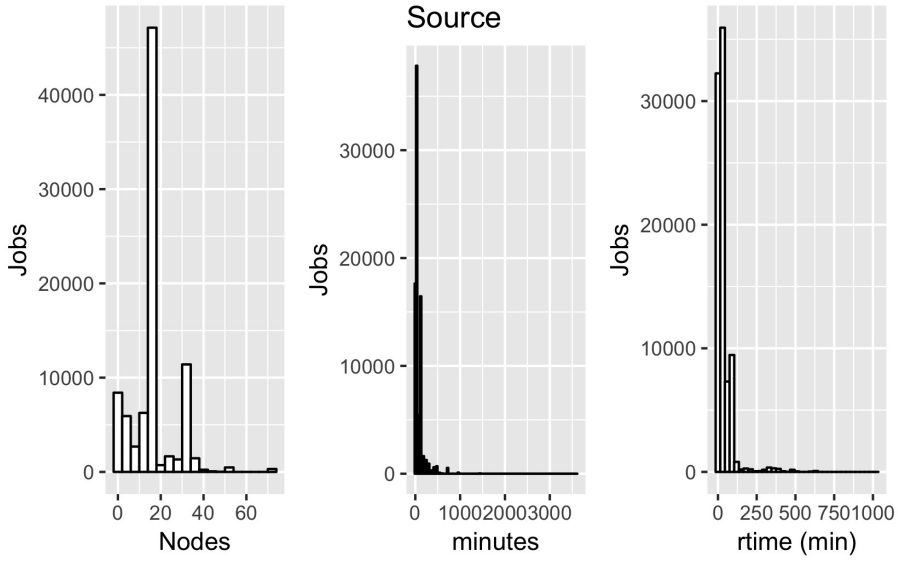

Figure 24: Distribution of source dataset.

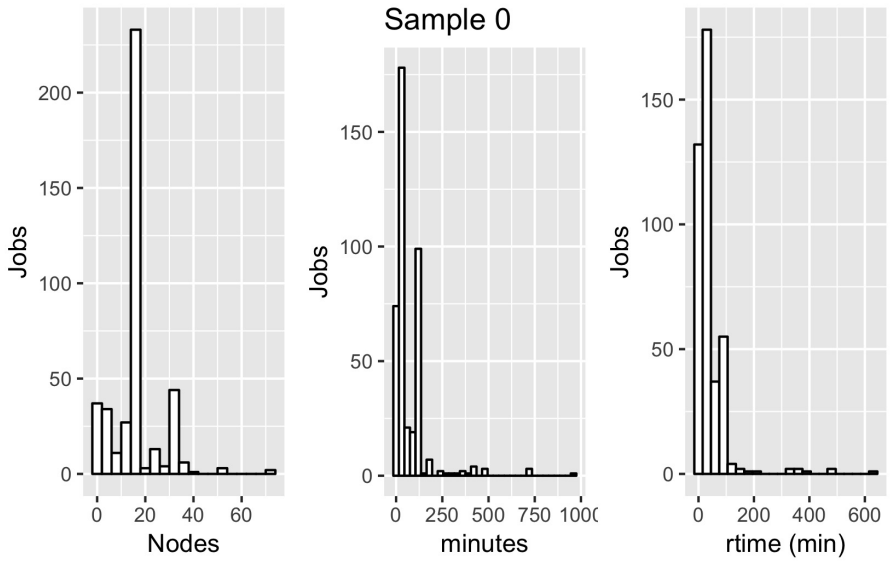

Figure 25: Distribution of sample 0.

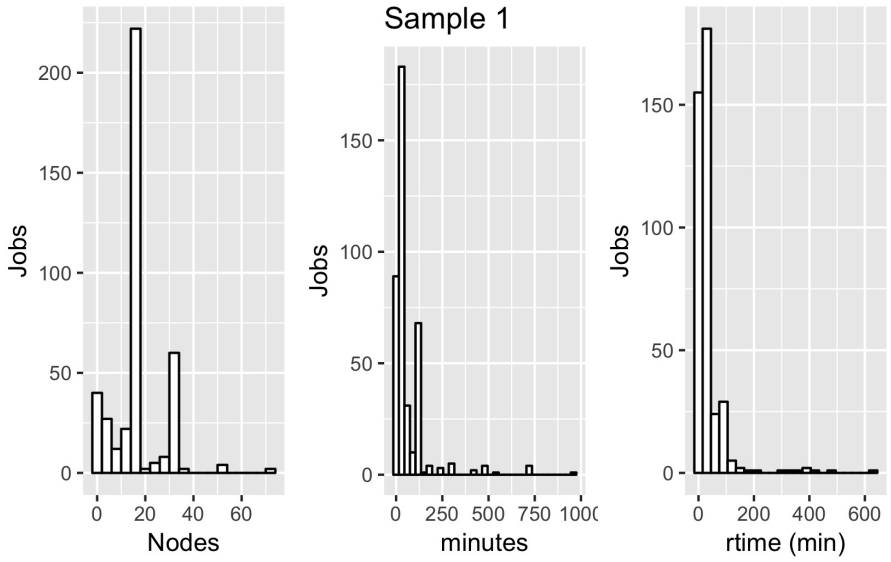

Figure 26: Distribution of sample 1.

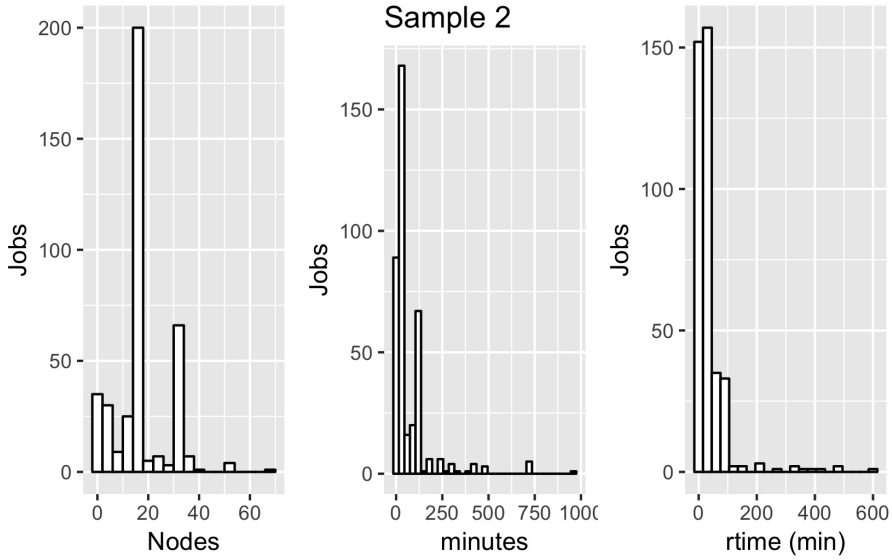

Figure 27: Distribution of sample 2.

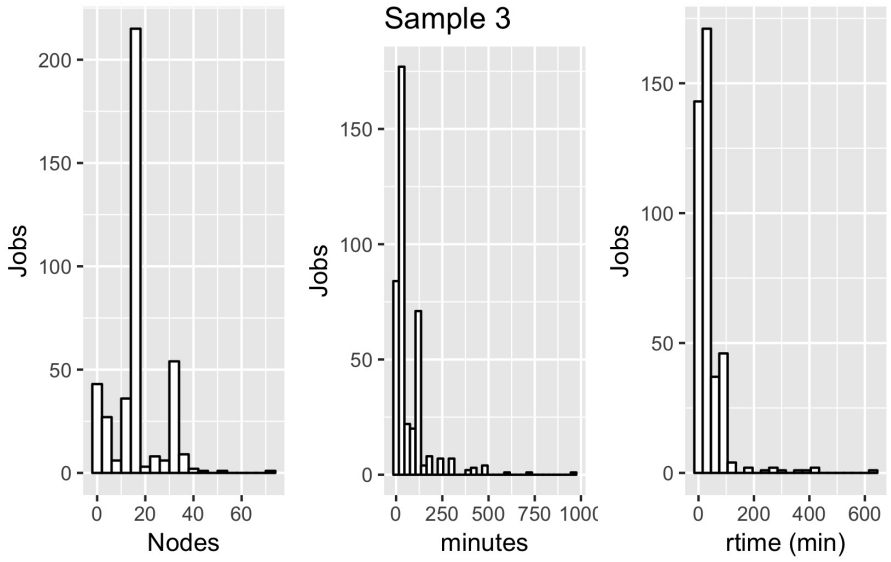

Figure 28: Distribution of sample 3.

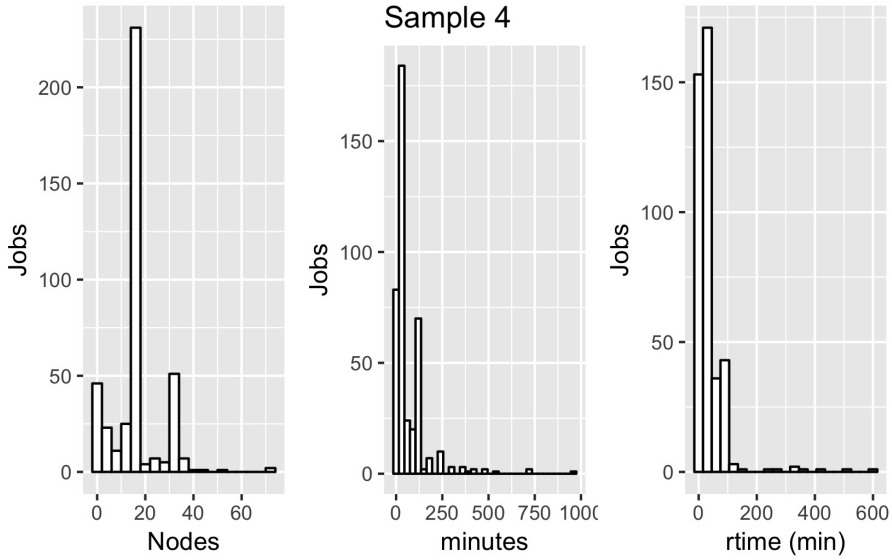

Figure 29: Distribution of sample 4.

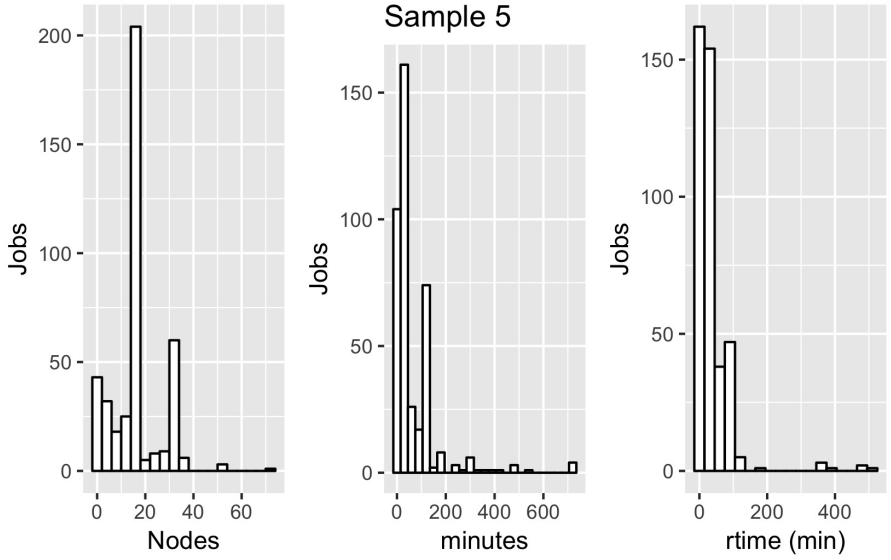

Figure 30: Distribution of sample 5.

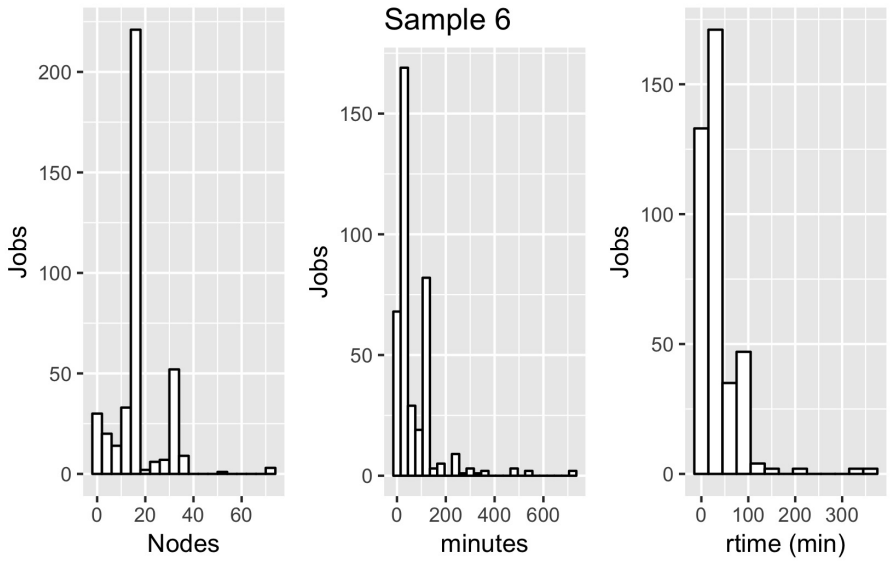

Figure 31: Distribution of sample 6.

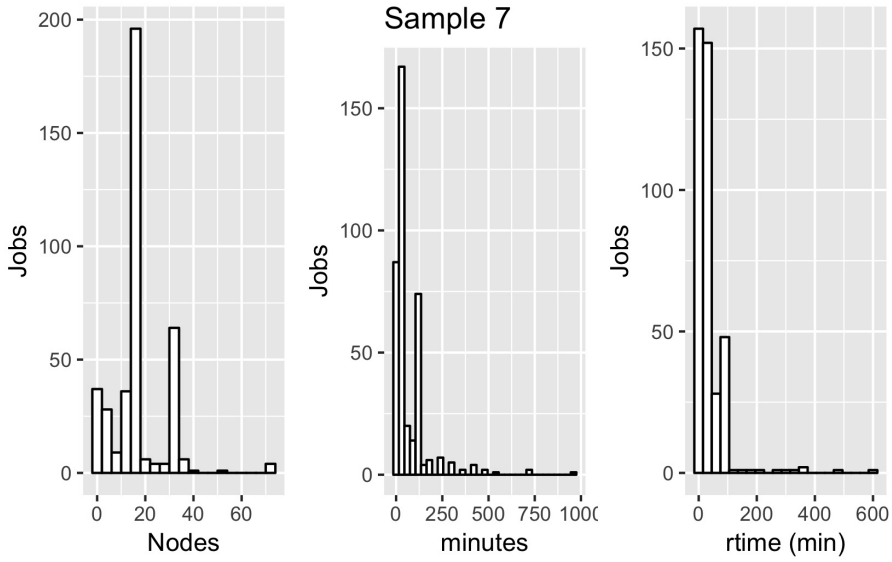

Figure 32: Distribution of sample 7.

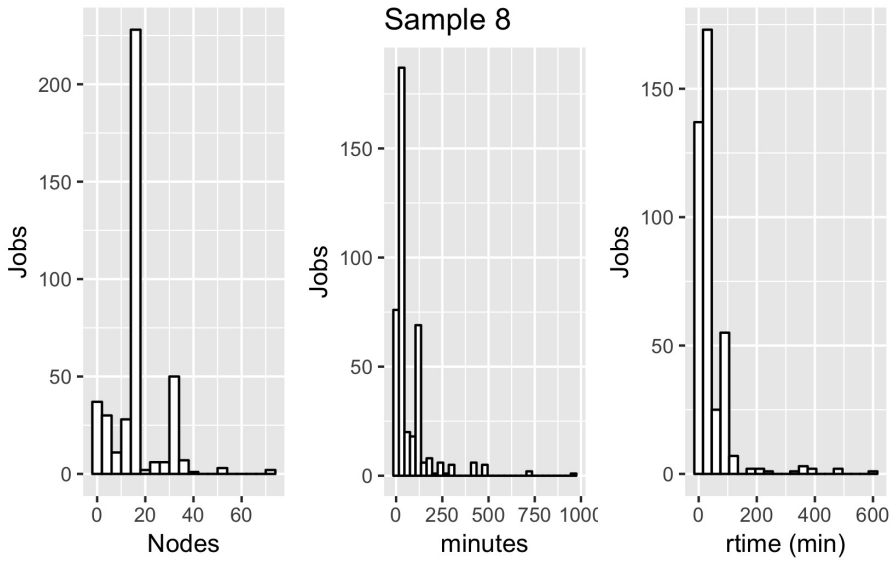

Figure 33: Distribution of sample 8.

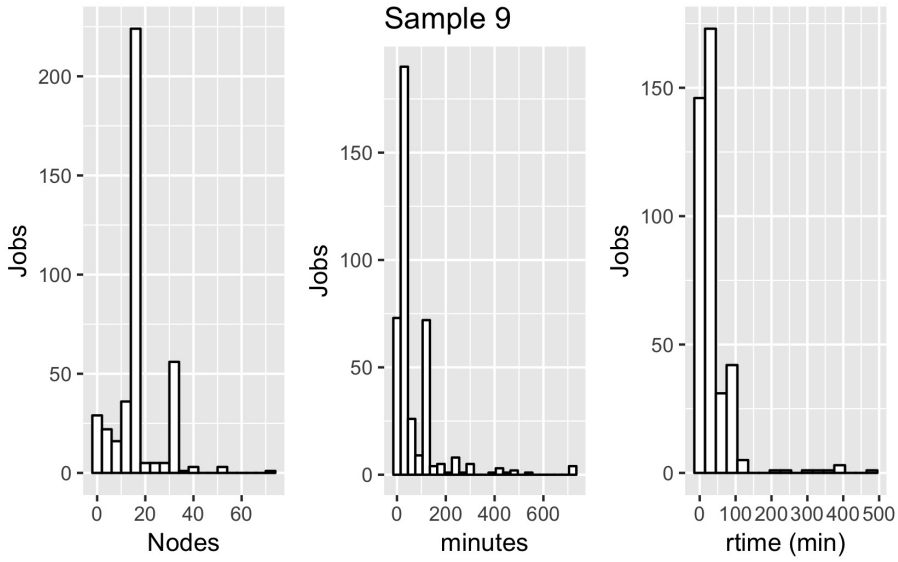

Figure 34: Distribution of sample 9.

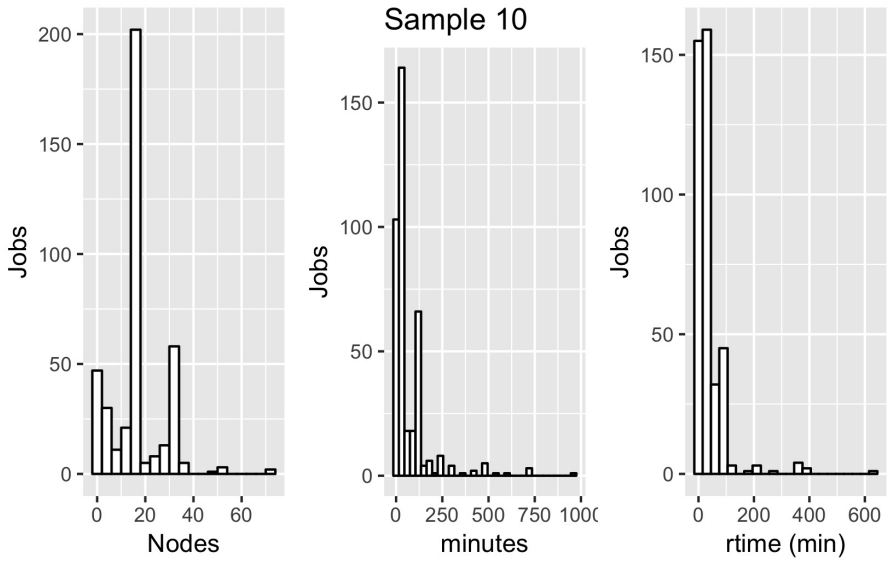

Figure 35: Distribution of sample 10.

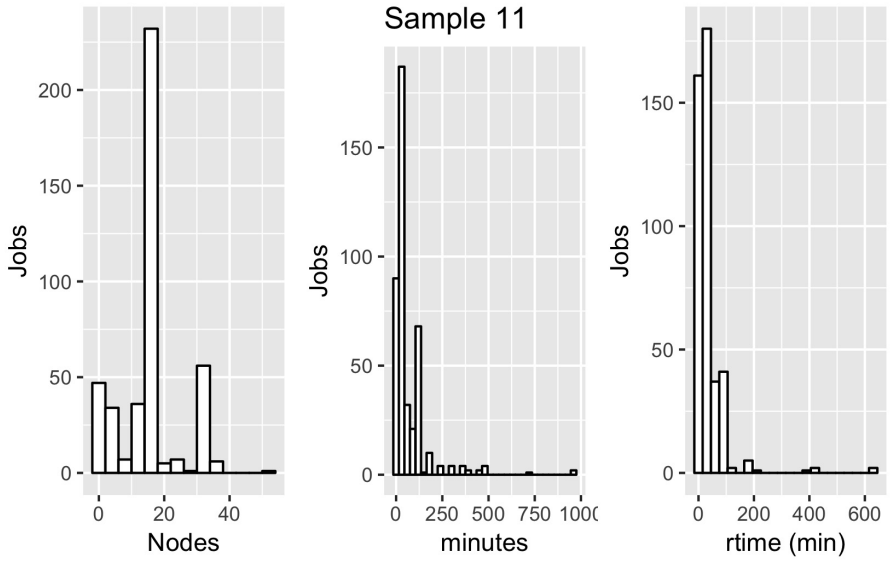

Figure 36: Distribution of sample 11.

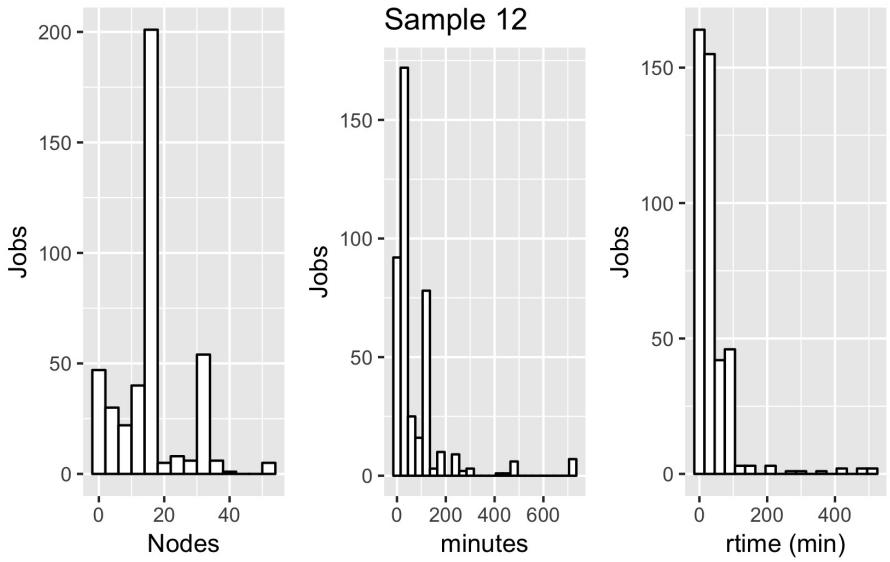

Figure 37: Distribution of sample 12.

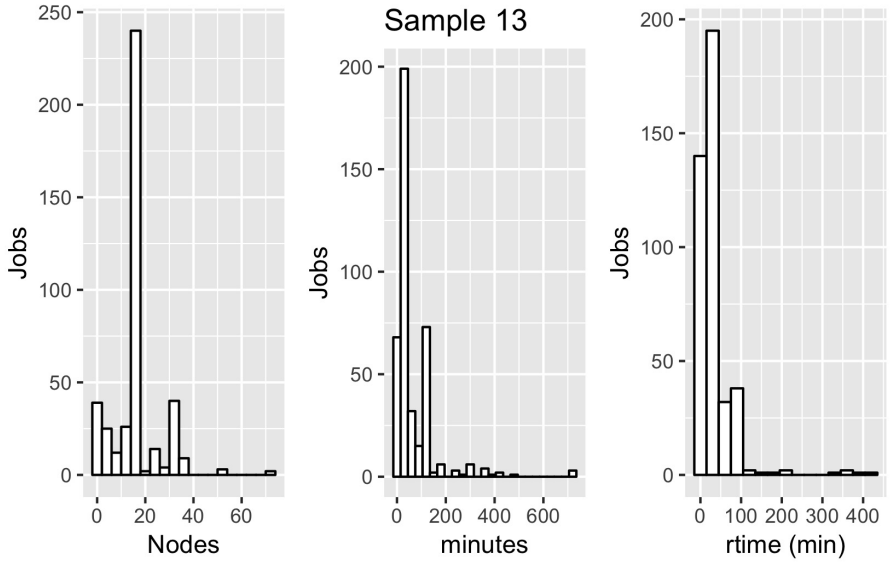

Figure 38: Distribution of sample 13.

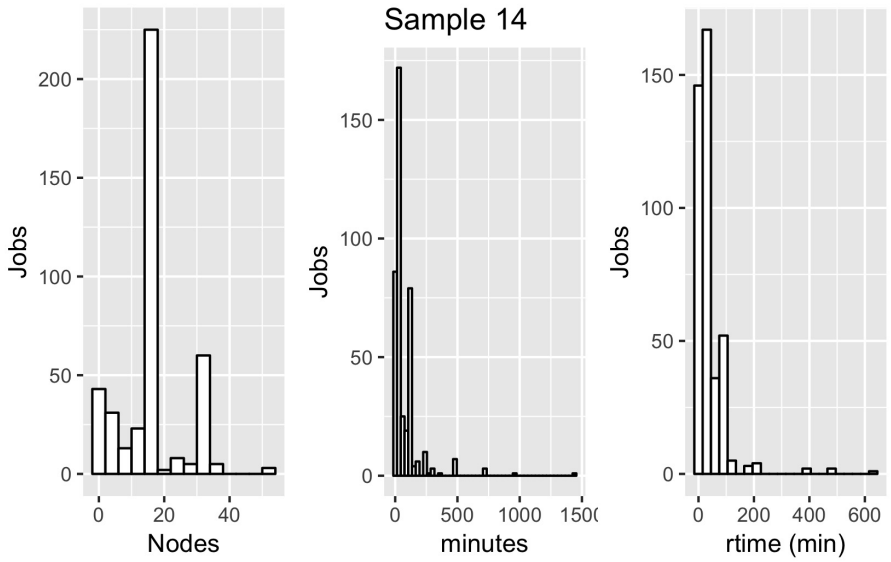

Figure 39: Distribution of sample 14.

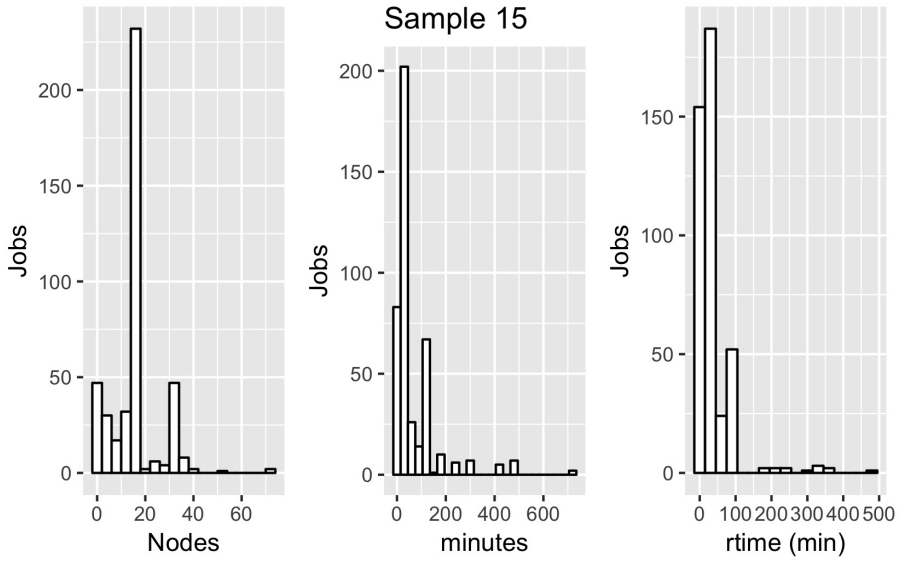

Figure 40: Distribution of sample 15.

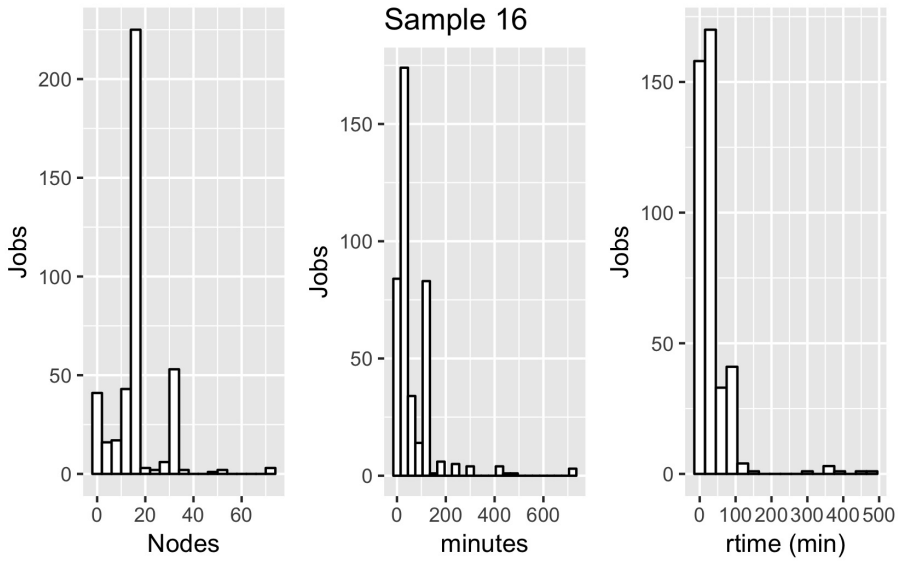

Figure 41: Distribution of sample 16.

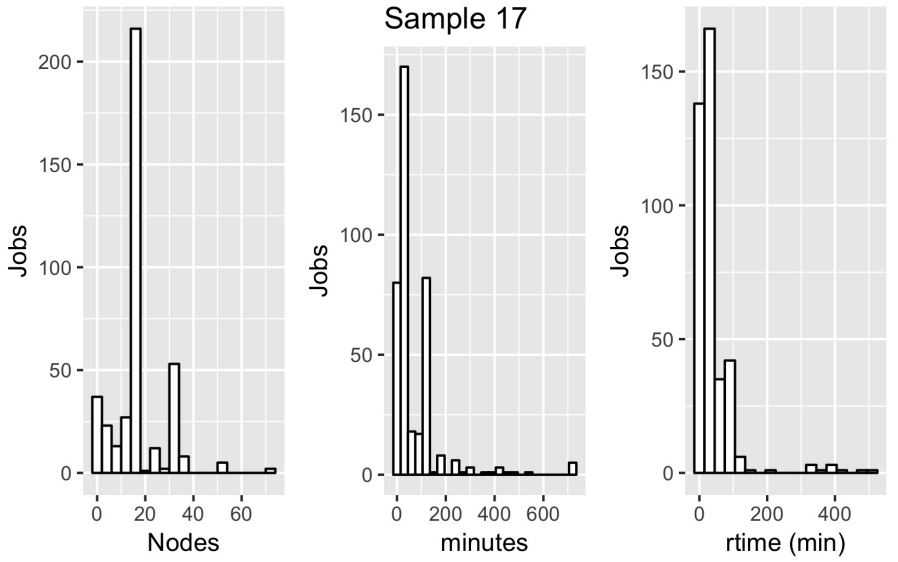

Figure 42: Distribution of sample 17.

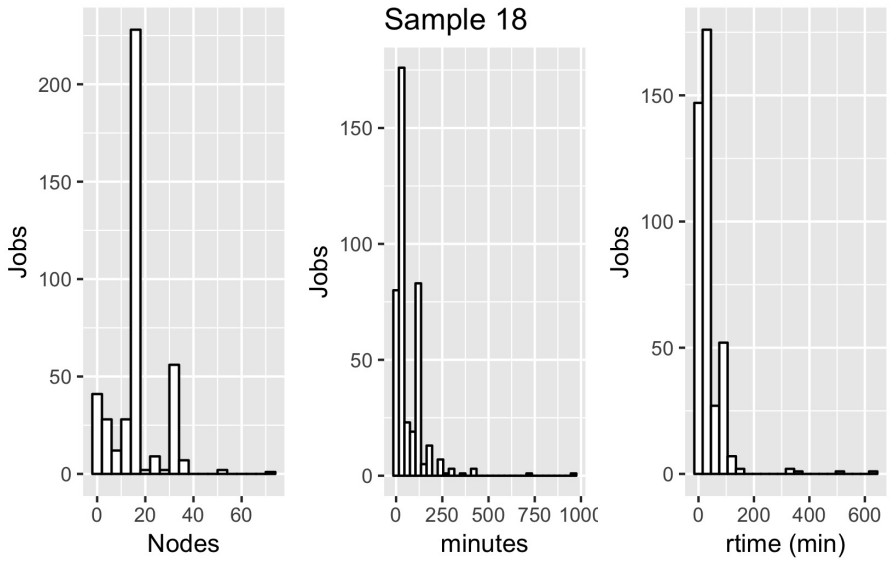

Figure 43: Distribution of sample 18.

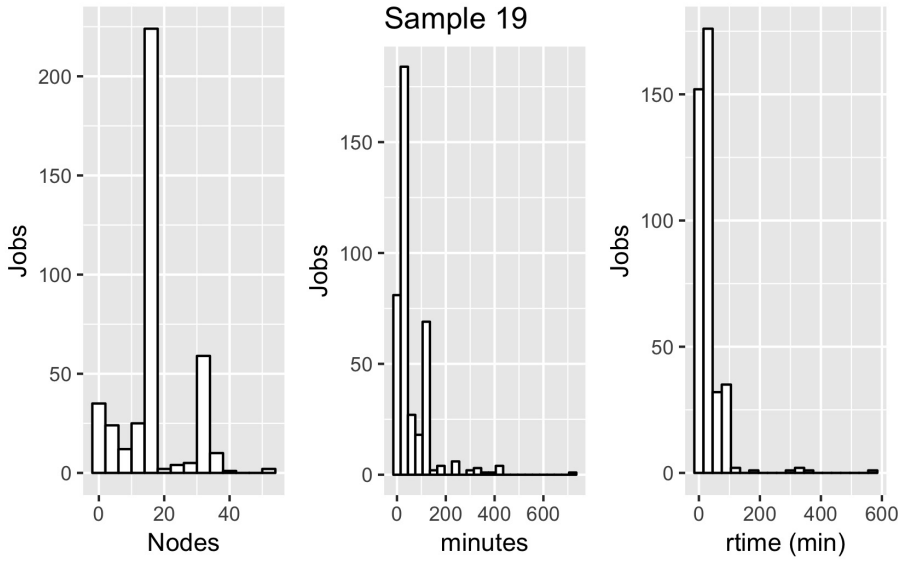

Figure 44: Distribution of sample 19.

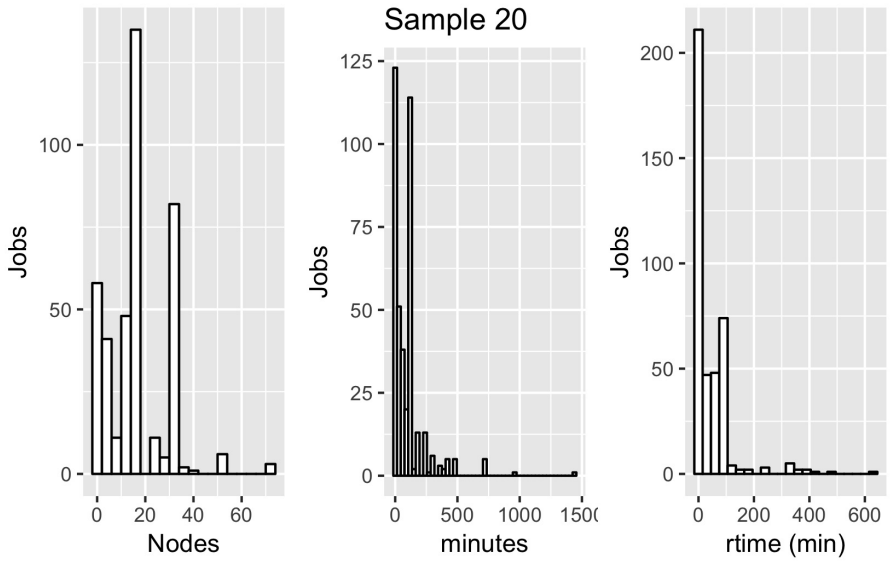

Figure 45: Distribution of sample 20. Excludes jobs with 40 nodes.