Facultad de Ciencias

IMPLEMENTACIÓN DE ALGORITMOS DE PLANIFICACIÓN PARA CLÚSTERS HETEROGÉNEOS
IMPLEMENTATION OF SCHEDULING ALGORITHMS FOR HETEROGENEOUS CLUSTERS

Trabajo de Fin de Grado
para acceder al

GRADO EN INGENIERÍA INFORMÁTICA

Autor: Alejandro Sánchez Sousa

Director: Esteban Stafford Fernandez

Junio - 2023

RESUMEN

En la actualidad existen numerosos problemas de gran escala computacional, que no son capaces de resolverse utilizando un único computador. Para hacer frente a este problema se ha desarrollado la computación con clústeres.

Un clúster es un conjunto de computadores o nodos que están conectados entre sí y trabajan juntos de manera coordinada para resolver un problema. Cada nodo o computador de un clúster se corresponde con una máquina distinta con unas determinadas prestaciones y que, por lo tanto, se comporta de manera diferente. Este conjunto de máquinas se conoce como clúster heterogéneo. Para la coordinación de los nodos, es necesario un software que distribuya toda la carga de trabajo entre los distintos computadores de la forma más eficiente posible.

El software encargado de distribuir el trabajo entre los nodos del clúster se llama planificador, y tomando como referencia un algoritmo de planificación, determina que trabajo es destinado a cada nodo y lo planifica. Existen una gran multitud de algoritmos de planificación que toman decisiones en función de distintas políticas y heurísticos.

El objetivo es elegir una serie de algoritmos de planificación e implementarlos en IRMaSim, un simulador desarrollado por la Universidad de Cantabria que es capaz de imitar el comportamiento de un clúster cuando le llega una carga de trabajo. Tras implementarlos, se llevarán a cabo simulaciones reales con cada uno de los planificadores y se compararán y evaluarán los resultados.

Palabras Clave: Clúster, nodo, heterogéneo, planificador, algoritmo de planificación, heurístico, simulador.

SUMMARY

Nowadays, there are numerous computationally intensive problems that cannot be faced using a single computer. To solve this issue, cluster computing has been developed.

A cluster is a set of computers or nodes that are connected and work together in a coordinate way in order to solve a problem. The majority of the real clusters are heterogeneous because each of the nodes or computers is a different machine with different features and level of performance. In order to coordinate properly all nodes, a piece of software that distributes all workload into the nodes in the most efficient way is needed.

This piece of software is known as workload Manager. It distributes the workload into the selected nodes by taking the results of a scheduling algorithm. It is not easy to decide which algorithm to use in order to schedule, as there is a wide variety of scheduling algorithms, each one based on a policy or heuristic.

The main purpose is to make a selection of different scheduling algorithms based on previous results and implement them in IRMaSim, a simulator developed by Universidad de Cantabria that imitates the behaviour of a cluster. Then execute some realistic simulations and finally, compare and evaluate the results obtained.

Keywords: Cluster, node, heterogeneous, workload manager, scheduling algorithm, heuristic, simulator.

Contents

1 Introduction

Cluster computing is a revolutionary approach in the field of computer science, that has transformed how large-scale computational problems are executed. They appeared due to the necessity of having more computing power than a single machine. So that more complex problems could be resolved in time. Nowadays, cluster computing is used in a wide variety of fields.

In the science and engineering environment, cluster computing is being used to perform complex simulations [9], like data analysis or computational modelling. In meteorology, clusters are used to execute high-resolution weather models and predict atmospheric behaviour with greater precision [2].

In the business realm, clusters take part in analysing big volumes of data, such as real-time transactions, big data or even business intelligence applications. For instance, electronic commerce companies use clustering in order to analyse client preferences and buying patterns in order to optimize customers' experience.

In the field of entertainment, clusters are used for rendering graphics and special effects in the film and video game industry. These applications require high computational performance to generate high-quality images and realistic visual effects. Clusters allow the graphics to be distributed among multiple nodes, speeding up the rendering process and significantly reducing the time required to complete complex audiovisual projects.

To sum up, cluster computing has revolutionized the way we approach large-scale computational problems. Its capabilities for parallel processing, scalability, and handling large volumes of data have driven significant advancements in various fields, from scientific research to entertainment and business decision-making. With ongoing technological advancements, clusters are expected to continue playing a crucial role in driving innovation and achieving more efficient and faster computational solutions.

Modern computer clusters are operated by research entities and serve a multitude of users. The users submit jobs to the cluster with the tasks they want to execute. A very important application in the cluster is the *workload manager*. It is in charge of receiving the jobs from the users and scheduling them to the appropriate computing resources in the cluster. This is a complex task in itself, as it is an instance of the job shop problem, which is a np-hard problem. To reach an acceptable solution to this problem, workload manager developers have resorted to heuristic algorithms. Through the use of different estimations, they are capable to reach suboptimal solutions in a reasonable amount of time. Furthermore, modern clusters are usually composed by computers of different performance, making them heterogeneous. Thus, the scheduling problem of the workload manager becomes significantly more complex in heterogeneous clusters.

The group of Computer Technology and Architecture of the Universidad de Cantabria has developed a simulator to study the workload manager algorithms in heterogeneous clusters. But this lacks the implementation of well known heuristic algorithms that would improve the scientific value of their research contributions. Therefore, in this work, several algorithms specific to scheduling jobs in heterogeneous clusters have been selected and implemented in the simulator.

The objectives of this work are:

- Review well known heuristic scheduling algorithms for heterogeneous clusters.
- Select a sample of relevant algorithms.
- Implement the selected algorithms in the simulator.
- Test and compare the performance of the implemented algorithms.

The remainder of this document is structured as follows. Chapter 2 covers some introductory concepts needed in the development of the document. Chapter 3 explains the new algorithms implemented in the simulator, and Chapter 4 sets some explanations referred to the implementation decisions. All the results and tests are in Chapter 5. Finally, Chapter 6 summarizes all work done and give some conclusions based on the experiments in 5.

2 Fundamentals

A computer cluster is basically a set of computers that work together as a single system in order to achieve a common objective. Clusters were created to reduce the execution time of large scientific applications. A key aspect in the operation of these systems is the adequate division of the workload into manageable portions that can be sent to each computer in the cluster. A piece of software, called the *workload manager* [7], is used to provide this important functionality of computer clusters.

However, it is common in modern clusters that not all the computers are equal in performance and speed. Such clusters are called heterogeneous, and the goal of the workload manager is more challenging than in homogeneous clusters. In general, the objective of the workload manager is to finish all the work in the shortest possible time.

2.1 Definitions

It is important to state some definitions for concepts that will be used throughout this work.

- **Node** refers to each of the computers that form a computing cluster. In the context of this work, each node has different performance, leading into heterogeneous clusters. Each node has a set of computing elements that are called cores.
- **Core** is a processing unit within a node that can independently execute instructions. Multiple cores in a node enable parallel processing, which can improve overall system performance by allowing simultaneous execution of different parts of a problem.
- **Job** is the way a user has to submit a computing problem to the cluster. It consists of a command line that launches the execution of the application, and some metadata. The latter can include, requested time, referring to the estimated execution time of the task, the requested number of cores, memory, etc. A job can launch one or more tasks.
- **Task** is a part of a job that can be executed on a separate core. In this work, all the tasks of a job must execute simultaneously and in the same node. Each task must execute in a single core on its own.

Even though clusters may consist of heterogeneous nodes, the nodes themselves are considered homogeneous. It is assumed that all cores within a node exhibit uniform performance. This assumption is based on the prevalence of modern machines that typically have processors operated for high performance across all cores. Situations involving architectures like ARM big.LITTLE [6] are not taken into account.

2.2 IRMaSim

The group of Computer Technology and Architecture of the Universidad de Cantabria has developed IRMaSim [1]. It is a simulator capable of simulating the behaviour of a heterogeneous cluster at a very high level of abstraction. This simulator can be used to study and evaluate the performance of scheduling algorithms. As this is a simulator, the goal is to predict the execution time of numerous jobs, that if run in a real system would require a long time, in a few seconds. This simulator considers that each task has a number of instructions that a processor will execute at a rate determined by its clock frequency and a few other factors. This way eliminating the overhead of modelling complex structures of the architecture of the processor, like speculative execution or cache memory. This level of abstraction is suitable for the study of scheduling algorithms.

It is important to describe how the simulator defines the workload, the cluster architecture and the workload manager, as the development of this work is heavily based upon them.

2.2.1 Workload

A workload refers to a JSON file in which a user specifies a set of jobs to be executed within a cluster, commonly named `workload.json`. Each job is defined by various parameters. It is noteworthy that IRMaSim allows the workload manager to see some of these parameters, while others are used exclusively for simulation purposes.

- **ID** is the identifier of a job. It must be unique in other to distinguish it from other jobs.
- **submit_time** is the time when the job is supposed to enter the cluster. Before this time, the workload manager does not know this job exists.
- **ntasks** corresponds to the number of tasks of a job.
- **req_time**is the maximum time a node can be executing the job. This value is used by the workload manager to improve the scheduling behaviour.
- **req_ops** as the number of instructions of each task.
- **opc** refers to the number of instructions of a job that a node can execute in one cycle.

2.2.2 Platform

The cluster architecture is known in IRMaSim as the platform. Like the workload, the platform is defined in a file, usually named `platform.json`, that enumerates all the parts and properties of the cluster. It is based on a dictionary structure. The three basic components that are defined are: *processor*, *node* and *cluster*.

- **Processor** refers to each type of processor that can be found in the nodes of the platform. They are defined by a name and ID. Their main parameters are: the number of cores, *cores*; the frequency, *clock_rate*; and the number of operations per cycle, *flops_per_cycle*. With these parameters, and knowing the number of instructions in a task, the simulator is capable of estimating the execution time.
- **node** is identified by its ID and contains a list of processors. In order to select a processor from the ones defined, its name and the number of units within the node must be specified. Nodes are homogeneous, meaning that there will not be different types of processors within the same node.
- **cluster** also has an ID and a list of nodes that, as it happens with the processors, have been defined. For each node included in the cluster, its name and the number of these type of nodes must be set. IRMaSim allows defining heterogeneous clusters by using different types of nodes in the same cluster.

The simulator takes all the resources in the *platform.json* file and builds the structure where the different jobs from the workload manager will be allocated.

2.2.3 Workload Manager

As mentioned in Chapter 1 a workload manager is a piece of software that distributes all work into the cluster. To be more precise, it takes all incoming jobs and distributes them among all nodes of the cluster, following a scheduling algorithm.

IRMaSim has already some workload managers with the implementation of some basic algorithms. The simplest algorithm is called **Minimal** and is based on the scheduling algorithm *OLB* [8]. From the entering jobs, *Minimal* takes the first one in the list and allocates it to the first node available to execute it.

Table 1: Classic policies exposed through the Action Space

Job Selection Policies	
random	Any job scheduling
first	Job with the earliest submit time
shortest	Job with the shortest user requested time
longest	Job with the highest user requested time
smallest	Job with the lowest user requested cores
low_mem	Job with the lowest user requested memory
low_mem_bw	Job with the lowest user requested memory bandwidth
Resource Selection Policies	
random	Any resource
high_gflops	Resource with the highest current FLOPS
high_core	Resource with most available cores
high_mem	Resource associated to the node with most memory
high_mem_bw	Resource associated to the processor with most memory bandwidth
low_power	Resource with the lowest current Watts

Heuristic is an implementation of various simple scheduling algorithms based on how jobs and nodes are ordered. In essence, the heuristic scheduler employs various approaches to order the list of incoming jobs and the list of nodes within a cluster. By combining these approaches, several basic schedulers can be implemented. Table 1 enumerates the these sorting policies. One such example is the random scheduler, which randomly selects a job and assigns it to a randomly chosen node within the cluster that can accommodate its execution.

Another scheduler that can be implemented is the MCT (Minimum Completion Time) scheduler [8]. In this approach, the jobs are ordered based on their submitting time with the *first* option, and nodes are ordered based on their number of FLOPS, *high_gflops* in descending order. The first job is assigned to the first available node, which is the fastest available one, resulting in the least Completion Time (CT). Subsequently, both the job and the node are removed, and the process is repeated with the next pair. Once all nodes all filled with jobs, the nodes list is filled again and the process is repeated.

Other meaningful algorithm will be the one that combine the *shortest* with the *high_gflops* policy. This algorithm is very similar to the *min-min* algorithm explained in Chapter 3. It orders jobs by their *requested_time* in ascending order and allocates each job into the node that has the *minimum_completion_time* for that node. Once it is allocated, the job is removed from the incoming list and the node is removed from the available nodes. In case there are more jobs than nodes, the list of available nodes is refilled when it is empty. This algorithm will be called *shortest_fastest*.

Very similar to the previous one will be the one resulting of combining the *longest* with the *high_gflops* policy. This algorithm behaves the same as *shortest_fastest*, but it orders the incoming jobs by its *requested_time* in descending order. It will be called *longest_fastest*, and it will schedule first the apparently longest job instead of the shortest one.

By employing these different ordering strategies, the heuristic scheduler allows for the implementation of diverse scheduling algorithms that can optimize job allocation within the cluster.

2.3 Execution

When the three main structures are generated, IRMaSim simulates the execution of the workload on the platform by advancing a variable amount of time and executes scheduling and allocating according to the workload manager.

The code in charge of all the simulation is *simulation.py* and implements all process in a while loop. This loop is controlled by *delta-time*, a variable that keeps track of the advanced time in the simulation. When the simulation time matches the submitting time of the next set of incoming jobs, the simulator calls the function *on_job_submission* of the workload manager that schedules all incoming jobs. When the simulation time reaches the finishing time of a job, *on_job_completion* of the workload manager is called, and the job is deallocated. This loop finishes properly when there are no more sets of incoming jobs and all incoming jobs have been executed.

However, there might be situations where the simulations will be finished without executing all jobs, so When IRMaSim is executed the state of the jobs is displayed.

- A job is in a future state when the job has not even arrived to the cluster. When $simulation\_time < submit\_time(job)$.
- A job is in the queued state when it has been already scheduled, but the node where it is allocated has not started its execution yet.
- A job is in executing state when the node where it is allocated has already started but have not finished its execution yet.
- A job is in finished state when it finishes its execution.

Other parameters are also displayed, however, as they are more related to the performance of each scheduling algorithm, they will be explained in Chapter 5.

2.4 Other Software

In addition to IRMaSim, this project utilized other software tools. Python open-source libraries such as *NumPy* and *pyplot* were employed to generate the graphs presented in Chapter 3 and 5. GitHub was utilized to manage and track the progress of the project. The IRMaSim software can be accessed at `https://github.com/Pepetillo300/IRMaSim`.

3 Design

The main objective of this work is to implement a variety of scheduling algorithms for heterogeneous clusters with different strategies and heuristics. For the research of the heterogenous cluster workload manager algorithms, the article by Tracy D. Braun, Howard Jay Siegel, and Noah Beck [8] has been studied. In it, the author describes the state of the art in terms of these algorithms. After seeing the results of the experiments presented there, it was decided to implement four algorithms. *min-min*, *max-Min*, *duplex* and *A**.

These algorithms will be compared using various workloads on different platforms to determine their performance in both general and specialized scenarios. The newly introduced algorithms are more complex and rely on more precise heuristics compared to the existing ones in IRMaSim.

3.1 Mathematical Definitions

- **Node:** From a set of nodes inside a cluster N of length $m(N)$, each node $\{n_k \,|\, 0 \leq k < m(N)\}$ constitutes a homogeneous structure where a job can be planned. It is homogeneous because all the cores that belong to a node $\{c_{k,p} \,|\, 0 \leq k < m(N),\, 0 \leq p < m(n_k)\}$ perform in the same way.
- **Job:** From a list of jobs J of length $m(J)$, each job $\{j_i \,|\, 0 \leq i < m(J)\}$ must be executed by a single node n_k. Additionally, their scheduling can be statically performed in various ways, allowing different mappings of the set of jobs J.
- **Task:** From a list of tasks per job T_i, of length $m(j_i)$, each task $\{t_{i,l} \,|\, 0 \leq i < m(J))\ , 0 \leq l < m(j_i)\}$ must be executed in the same node at the same time. As, there are some pieces of data or code shared by all tasks. So that each task is allocated into a core.

3.2 Time Metrics

These definitions are base on the ones given in [8].

- **Expected Time to Compute** $(ETC(j_i, n_k))$ is defined as the time a job j_i is executed by a node n_k, without considering previous or following executions of other jobs. As nodes are homogeneous, the ETC of a job j_1 is the same as the ETC of its tasks.

$$\{ETC(t_{i,0},\, n_k)\} = \{ETC(t_{i,1},\, n_k)\} = \{ETC(j_i, n_k)\}.$$

- **Machine availability time** $Mat(n_k)$ is defined as the the earliest time that a node n_k can complete the execution of all the jobs that have previously been assigned to it. As nodes in the platform contain different times, the *Machine availability time* of a node n_k is equal to the maximum Mat of its cores:

$$mat(n_k) = \{max_{0 \leq p < m(n_k)}(mat(c_{k,p}))\}$$

- **Completion Time** $ct(j_i, n_k)$ is the *Machine availability time* of the selected node added to the *Expected Time to Compute* of job j_i when scheduled to the node n_k [8].

$$ct(j_i, n_k) = mat(n_k) + ETC(j_i, n_k)$$

3.3 Min-Min

The *min-min* algorithm schedules jobs based on their anticipated execution time. The job with the shortest expected execution time is assigned to the node that can execute it first. This process is repeated for all incoming jobs.

3.3.1 Definition

Min-min is a heuristic algorithm that takes as a parameter the set of all incoming jobs. $J = \{j_0, ..., j_i, ..., j_{m(J)-1}\}$. For each job $\{j_i \in J\}$ its completion time for each one of the nodes in the cluster is estimated $(ct(j_i, n_k \mid 0 \leq k < m(N))$ and its minimum value is selected.

$$M_{j_i} = \{min_{0 \leq k < m(N))}(ct(j_i, n_k))\}.$$

Afterwards, the job with the minimum completion time M is selected and assigned to its corresponding machine. This process is repeated with all the jobs until there are no jobs to be scheduled. After scheduling a job to a node n, the machine availability time of the node is updated with the execution time of the job j.

$$mat(n_k) = mat(n_k) + ETC(j_i, n_k)$$

3.3.2 Pseudocode

A pseudocode has been developed by taking inspiration of algorithm E in [5].

Algorithm 1 Min-Min

```
1: Order jobs by their $requested\_time$
2: for $job = j_0, j_1, j_2, \ldots j_{m(J)-1}$ do
3:     for $node = n_0, n_1, n_2, \ldots n_{m(N)-1}$ do
4:         if $j_i$ does not enter $n_k$: then
5:             Jump to next iteration
6:         end if
7:         $ct(j_i, n_k) = mat(n_k) + ETC(j_i, n_k)$
8:     end for
9:     $M_{j_i} = \{min_{0 \leq k < m(N))}(ct(j_i, n_k))\}$
10:    $n = \{n_{0 \leq k < m(N))} \mid ct(j_i, n_k) = M_{j_i}\}$
11:    $mat(n) = mat(n) + ETC(j_i, n)$
12:    schedule $j_i$ in node $n$
13: end for
14: Return Scheduling
```

The definition of *min-min* suggests that, for each job, its M_{j_i} must be computed, then, the job is assigned to its node and then, it is removed from its list. However, if nodes have enough cores, there may be cases where more than one job is assigned to the same node. Moreover, assigning a job to a node may not update the *machine availability time* of that node. So that, the heuristic M should be computed differently.

Min-min knows what is the *requested time* of a job so in pseudocode 3.3.2, all the jobs are ordered by this parameter, which is proportional to the number of operations and to the *ETC* of the slowest node.

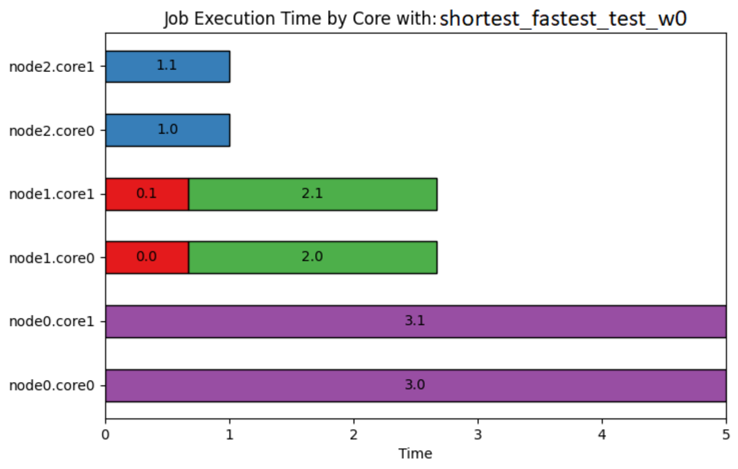
$$req\_time(j_i) = ETC(j_i, n_k \mid min_{0 \leq k < m(N))} mops\_per\_core(n_k))$$

Then, *min-Min* iterates over all the jobs, and for each job iterates over all the nodes and computes, if possible, each value of *completion time*. The *completion time* of a node is the maximum value of the completion time of its cores.

$$ct(j_i, n_k) = \{max_{0 \leq l < m(n)}(ct(t_{i,k}, c_{j,l})\}$$

Once the iteration over all the nodes is finished, its M is computed. Afterwards, n, which is the node whose *completion time* for j_i is equal to M is obtained, and its *machine availability time* is updated. Finally, the job is scheduled.

Min-min finds the lowest M of all the jobs and allocates it dynamically, as well as letting other jobs to be executed in the same node, if it continues to have the minimum *machine availability time*.



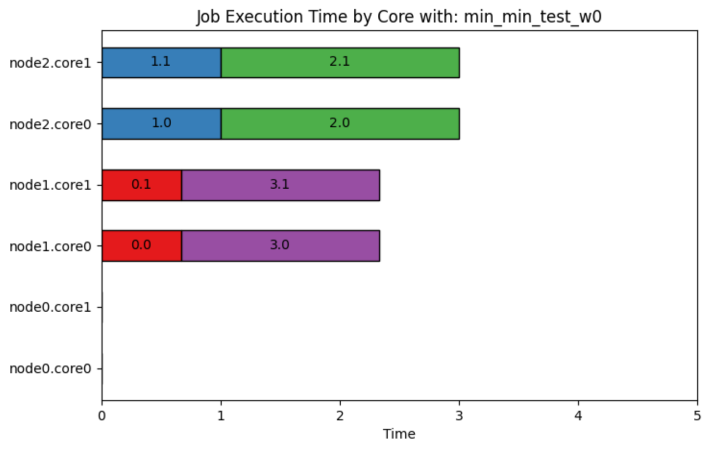


Figure 1: Comparison between the heuristic shortest jobs selection with high_gflops node selection and the Min-Min algorithm.

Figure 1 illustrates the distinction between the *min-min* and the *shortest_fastest* algorithms. Both algorithms start in the same way. They have their list of jobs ordered, and start by scheduling job 0, which is the shortest one. Node 1 and 2 are the fastest ones, and they perform the same, so, as there are no previous jobs scheduled, job 0 could be scheduled in any of them. In both cases, node 1 is selected. Then job 1 is scheduled in node 2 because, as it was established, it performs the same as node 1. There is not any difference between the two algorithms so far. However, in the heuristic algorithm, the job with ID 3 is assigned to node 0 based on the anticipation that this node will be the quickest since the other nodes already have scheduled jobs. On the other hand, *min-min* examines the completion times across all three nodes and determines that node 1 will be the swiftest for executing the job, so *min-min* schedules it there. Despite job 3 having to wait to commence its execution, it concludes earlier on node 1, suggesting that *min-min* provides a more accurate estimation. The same process occurs when scheduling job 2. Both schedule it in the node with the minimum *completion time*. As the nodes have already been selected, the possible nodes list in heuristic has been restarted. *Min-min* does not take into account which nodes have already been used, job 2 is simply scheduled in the node with the minimum *completion time*, in this case node 1.

3.4 Max-Min

The *max-min* algorithm arranges jobs according to their predicted execution time. It assigns the job with the longest estimated execution time to the node that is capable of executing it first. This procedure is iterated for all incoming jobs. Furthermore, the *max-min* algorithm takes into account both the jobs presently being executed on nodes and the jobs that are yet to be executed.

3.4.1 Definition

Max-min is a heuristic algorithm that takes as a parameter the set of all unmapped jobs. $J = \{j_0, ..., j_i, ..., j_{m(J)-1} \mid 0 \leq i < m(J)\}$. For each job, $\{j_i \in J\}$ its *completion time* for each one of the nodes in the cluster is estimated $ct(j_i, n_k \mid 0 \leq k < m(N))$ and its minimum value is selected.

$$M_i = \{min_{0 \leq k < m(N)}(ct(j_i, n_k))\}.$$

Afterwards, the job with the maximum *completion time* M is selected and assigned to its corresponding machine. This process is repeated with all the jobs until there are no jobs to be scheduled.

After scheduling a job into a node, the *machine availability time* of the node is updated with the execution time of the job *ji* in the node *nk*.

$$mat(n_k) = mat(n_k) + ETC(j_i, n_k)$$

3.4.2 Pseudocode

Pseudocode 3.4.2 has been developed by taking inspiration of algorithm D in [5]. However, due to the resources IRMaSim offers, there are some changes that will be explained.

Algorithm 2 Max-Min

```
1: Order jobs by their requested_time
2: for $job = j_{m(J)-1}, j_{m(J)-2}, j_{m(J)-3}, \dots j_0$ do
3:     for $node = n_0, n_1, n_2, \dots n_{m(N)-1}$ do
4:         if $j_i$ does not enter $n_k$: then
5:             Depreciate $n_k$
6:             Jump to next iteration
7:         end if
8:         $ct(j_i, n_k) = mat(n_k) + ETC(j_i, n_k)$
9:     end for
10:    $M_{j_i} = \{min_{0 \leq k < m(N)}(ct(j_i, n_k))\}$
11:    $n = \{n_{0 \leq k < m(N)} \mid ct(j_i, n_k) = M_{j_i}\}$
12:    $mat(n) = mat(n) + ETC(j_i, n)$
13:    schedule $j_i$ in node $n$
14: end for
15: Return Scheduling
```

The explanations and pseudocodes for both the *min-min* and *max-min* algorithms share similarities. However, the key distinction lies in the selection of jobs. While the *min-min* algorithm chooses the job with the *minimum completion time*, the *max-min* algorithm selects the job with the *maximum completion time*.

As it was done with the *min-min* algorithm, *max-min* firstly orders the jobs by their requested time. However, the list of incoming jobs is iterated backwards. For each job, all the possible nodes where it can be allocated are iterated and its M is computed. Then, the *machine availability time* of the selected node is updated, and the job is scheduled.

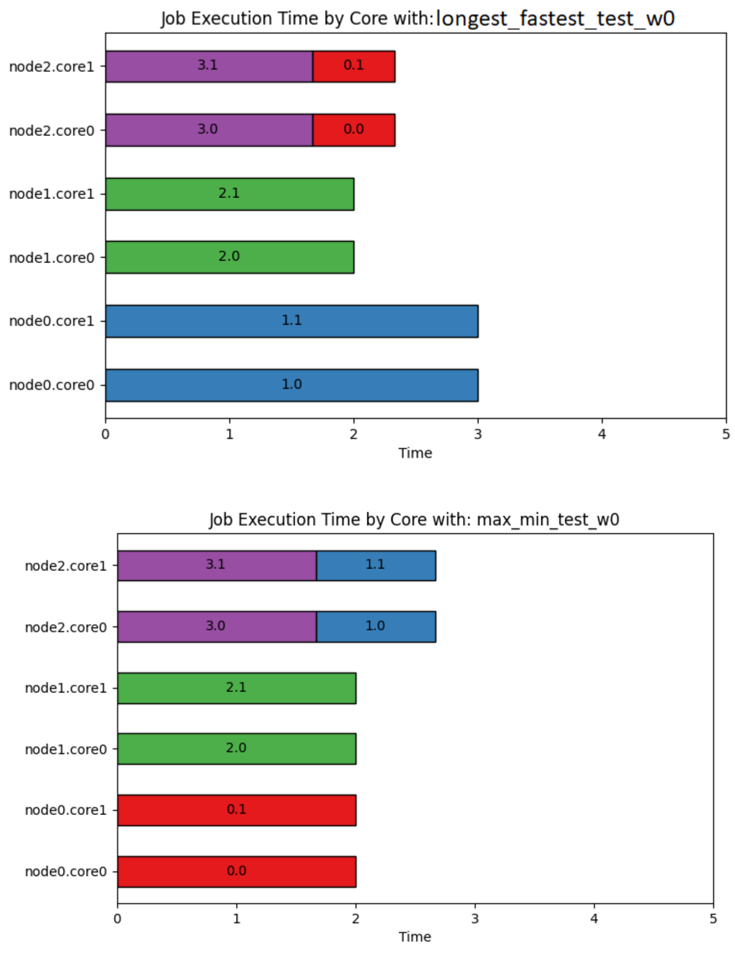


Figure 2: Comparison between the heuristic longest jobs selection with high_gflops node selection and the max-min algorithm

In Figure 2, the *max-min* algorithm is compared with the heuristic *longest high_gflops* algorithm. Both algorithms start by taking the heaviest job, job 2, and they schedule it in the fastest node, which is node 1. The workload and the platform are the same as the example in *max-min* so both node 2 and node 1 perform the same. The next job to schedule is job 3, both algorithms determine that node 2 will be the one with the lowest *completion time*, so job 3 is scheduled in node one in both cases. There are no differences so far, however, when scheduling job 1, the list of available node is the heuristic algorithm only contains node 0, as node 1 and 2 have already been used. That is why job 1 is scheduled in node 0. *max-min* schedules in the node with the least *completion time*, which is node 2. Finally, job 0 must be scheduled, in both the algorithms will be scheduled in the node with the least *completion time*, however considering that in *max-min* node 0 has not been used yet with the addition that job 0 is short enough to avoid increasing the *completion time* of the cluster, allows *max-min* to schedule job 0 in node 0. The heuristic algorithm has already used all nodes, so the one with the minimum *completion time* is node 0.

3.5 Duplex

3.5.1 Definition

Duplex is based on the combination of the *min-min* and *max-min* algorithms. For a list of jobs J that arrive to the cluster at a certain time, *duplex* estimates $ct(J, N)$, which is the completion time of the set of jobs in the cluster for *min-min* and *max-min*. This value is computed by taking the maximum *machine availability time* of the nodes in the cluster when all jobs are executed.

$$ct(J, N) = \{max_{0 \leq k < m(N)} mat(n_k)\}$$

Then *duplex* uses the better solution by taking the minimum *completion time* to schedule the jobs.

3.5.2 Pseudocode

Algorithm 3 Duplex

```
1: $ct(MinMin) \leftarrow MinMin()$
2: $ct(MaxMin) \leftarrow MaxMin()$
3: if $ct(MinMin) \leq ct(MaxMin)$ then
4:     Schedule as indicated in Min-Min
5: else
6:     Schedule as indicated in Max-Min.
7: end if
```

The pseudocode for the *duplex* algorithm is straightforward. It essentially implements the scheduling strategy of the algorithm that returns the *minimum completion time*. *Duplex* can effectively leverage the efficiency of both *min-min* and *max-min* under specific conditions. One advantage is that it can adapt its behaviour for each set of jobs J within the same simulation. For instance, for jobs entering at time 0, scheduling can be based on *min-min*, while for jobs arriving at time 10, scheduling can be based on *max-min*. However, a notable drawback is the overhead it introduces during execution. As mentioned in [3], both *min-min* and *max-min* have a temporal complexity of $O(mn^2)$.

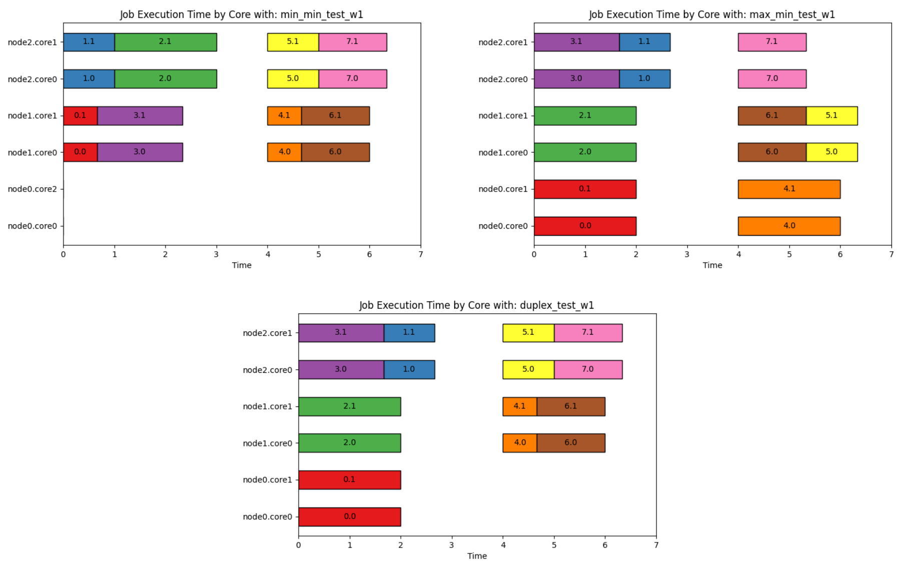


Figure 3: Comparison between Min-Min, Max-Min and Duplex algorithms

In the simulation shown in Figure 3, two queues of jobs have been launched to the cluster. One at time 0 and the other one at time four. For the first queue, the same jobs as the previous examples have been used. As *max-min* scheduling is better, duplex also uses it. Then the second query of jobs arrives. However, *min-min* is now better. So *duplex* takes its implementation.

3.6 A*

*A** is a complex heuristic based on an u-nary tree [8]. The tree represents the possible mappings of the set of jobs J in the cluster N.

A tree graph is an undirected, connected, acyclic graph consisting of vertices and edges. Trees are often represented visually with a diagram that branches out from a single vertex called the root. The remaining vertices are connected to the root and each other through edges, forming a hierarchical structure

3.6.1 Definition

The algorithm starts by defining the root vertex as an empty mapping, where no job has been assigned to any node yet.

It continues by adding children to the root vertex. The new vertices are marked as active, while their parent, the root, is marked as inactive. Each of these child vertices represents a mapping of the first job to each node in the cluster.

In order to go over the tree, the next vertex will be selected by comparing the values of $f(v)$ of each leave vertex. $F(v)$ is a lower bound estimation on the $ct(J, N)$ and the partial mapping that the vertex represents [3]. So that:

$$f(v) = g(v) + h(v)$$

On the one hand, $g(v)$ represents the *completion time* of the partial mapping of the vertex. $ct(\{j_0, j_1, j_2, \ldots, j_i\}, N)$. As defined in *duplex*, the *completion time* of a set of jobs is the value of the maximum *machine availability time* from all the nodes so that.

$$ct(\{j_0, j_1, j_2, \ldots, j_i\}, N) = \{max_{0 \leq k < m(N)} mat(n_k), \text{ when executed } j_i\}$$

On the other hand, $h(v)$ is a heuristic estimation on the difference between $g(v)$ and the *completion time* of the set of jobs J. The function $h(v)$ is defined with the use of two different functions, $h_1(v)$ and $h_2(v)$.

Let's define $h_1(v)$ as the maximum possible value between the current *machine availability time* and the *minimum completion time* for all the jobs that have not been mapped yet. So that $h_1(v)$ can be formulated as:

$$h_1(v) = max(0, (mmct(v) - g(v)))$$

With *mmct* being the maximum minimum completion time (the same value as the chosen in *max-min* algorithm), and with the assumption every node can allocate every job, an estimation of the completion time until all remaining jobs from J are executed.

The function $h2(v)$ calculates the projected increase in the completion time for the set of jobs J. It provides an estimate of the additional time needed for achieving the best possible solution for the remaining unmapped jobs, based on the current partial solution. This estimation assumes an ideal scenario where every job can be assigned to any node without any limitations or conflicts, resulting in an optimal outcome.

$$h_2(v) = max(0, (smet(v) - sdma(v))/\mu)$$

Once defined $h_2(v)$, *smet(v)* and *sdma(v)* will be defined too. The function *sdma(v)* is defined as the best *expected time to compute* for the remaining jobs in the cluster N.

$$smet(v) = \sum_{j=j_i}^{m(J)} (min_{0 \leq k < m(N)}(ETC(j, n_k)))$$

The function $sdma(v)$ quantifies the time available across all nodes that can be utilized for scheduling the remaining jobs without extending the *machine availability time* of the partial solution.

$$sdma(v) = \sum_{k=0}^{m(N)} (g(v) - mat(n_i))$$

When $f(v)$ has been computed for every children vertex, the leave node of the tree with the minimum $f(v)$ value is selected, $m(N)$ new vertexes are generated, and the process is repeated. However, there are cases where the selected vertex represents a less advanced mapping than the one from the current one. This factor combined with the explanation given before that every vertex can generate $m(N)$ children leads into pruning the tree when this is big enough. Pruning will be performed when adding a new vertex by selecting the leave node with the maximum $f(v)$ and deactivating it. Otherwise, *A** will lead into an exhaustive search.

3.6.2 Pseudocode

Algorithm 4 A*

```
1: $tree \leftarrow new\ Tree()$
2: $root \leftarrow new\ Tree\_node()$
3: $f(root) = 0$
4: $tree = tree + root$
5: parent = root
6: while $parent\_mapping \neq total\_mapping$ do
7:     $j \leftarrow next\_job\_to\_map$
8:     for $node = n_0, n_1, n_2, \ldots, nk, \ldots, n_{m(N)-1}$ do
9:         if $j$ does not enter $n_k$: then
10:            Jump to next iteration
11:        end if
12:        if $tree\_nodes \geq max\_tree\_nodes$ then
13:            $tree\_node \leftarrow \{leave\_node \mid max(f(v))\}$
14:            prune(tree_node)
15:        end if
16:        $new\_tree\_node \leftarrow new\ Tree\_node()$
17:        $f(new\_tree\_node) = compute\_fv(new\_tree\_node)$
18:        $parent.children() += new\_tree\_node$
19:        $tree\_nodes += 1$
20:    end for
21:    $parent \leftarrow \{leave\_node \mid min(f(v))\}$
22: end while
23: $scheduling \leftarrow scheduling(parent)$
24: return Scheduling
```

The pseudocode 3.6.2 follows closely the given explanation of the algorithm. Firstly, the tree structure is created and the *root* vertex, whose $f(v)$ is equal to 0 in order to be initialized to some value, is added. The $f(v)$ value of the *root* vertex is not relevant because it is the only possible vertex to choose in the first iteration because no other ones have been created yet.

Once *root* is added to the tree, the current job that *A** is going to map is selected in random order. For each job all the nodes in the cluster are iterated and firstly $A*$ checks if the job j can be allocated in node n_k. If it is not possible, the algorithm jumps to the next iteration, so that, no child for that possible mapping is created, being $m(N)$ the maximum number of children vertexes that can be generated by the same parent.

If the mapping is possible, $A*$ checks if it has to start pruning. As explained before, the vertex with the maximum value of $f(v)$ is pruned. Then, the vertex representing the current mapping is created and its $f(v)$ is computed. Afterwards, it is added to the tree. Once a vertex with the mapping of all jobs is obtained, the algorithm returns a scheduling following that mapping.

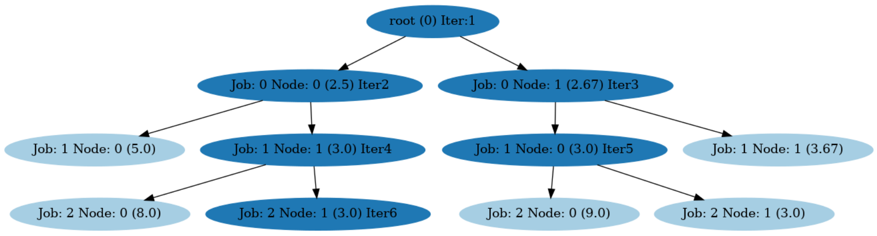

root (0) Iter:1
Job: 0 Node: 0 (2.5) Iter2
Job: 0 Node: 1 (2.67) Iter3
Job: 1 Node: 0 (5.0)
Job: 1 Node: 1 (3.0) Iter4
Job: 1 Node: 0 (3.0) Iter5
Job: 1 Node: 1 (3.67)
Job: 2 Node: 0 (8.0)
Job: 2 Node: 1 (3.0) Iter6
Job: 2 Node: 0 (9.0)
Job: 2 Node: 1 (3.0)

Figure 4: Example of execution of A*Algorithm

In order to show how $A*$ algorithm works, both the workload and the platform used in this example are smaller. The platform has only 2 nodes, node 1, which is the fastest one and node 0, which is the slowest. On the other hand, the workload has also been reduced from four jobs to three. In Figure 4 $A*$ starts in the root vertex and then computes its two children. *Job: 0, Node: 0* with $f(v) = 2.5$ has the smallest value, so this will be the next parent. *Job: 0, Node: 0* generates its children and $A*$ compares all $f(v)$ of all leaves of the tree so *Job: 0 Node: 1* with $f(v) = 2.67$ is chosen. This process is repeated until *Job: 2 Node: 1* with $f(v) = 3$ is chosen, as this node represents a complete allocation of the jobs the algorithm stops, and schedules them as indicated.

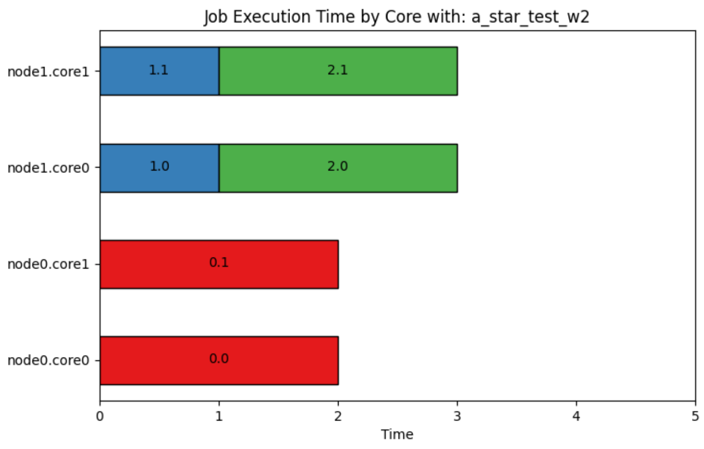


Figure 5: A*Algorithm Results

Once they have been simulated, it is seen in Figure 5 that the *completion time* of the workload J and the $f(v)$ estimated happen to be the same. To finish with, as the given example is actually small, there are not many vertexes generated, so that the tree has not been pruned.

4 Implementation

The main objective of this work was to implement the previous algorithms in IRMaSim., which is written in Python. This section covers the technical decisions and implementation details of the four algorithms

4.1 Implementation of Common Sections

Firstly, the two main aspects that are shared by all algorithms in IRMaSim will be commented. These are the structure of the functions of the workload manager as well as the global variables.

4.1.1 Common Variables

- **self.speedup** defines in a list each value of the speed-up of each node in relation to the slowest one. In order to compute it, the operations_per_second of each node is taken.
- **self.base_sp_time** is the number of operations_per_second the slowest node in the platform has.
- **self.scheduling** is a data structure made up by a list of lists of jobs. The bigger list contains smaller ones that represent each of the nodes in the platform. These smaller lists are filled up with the jobs that must be scheduled in that node. The attribute *self.scheduling* does not indicate the scheduling of the set of entering jobs J, but the scheduling of the simulation, $self.scheduling = scheduling(S)$.
- **self.cores_times** is a data structure that, as self.scheduling, consists of a list of lists where each small list represents a node. However, they are filled by float values that represent the $mat(c_{k,p}$ from $0 \leq p < m(n_k))$, the *maximum availability time* of each core.
- **self.running_jobs** is a list that contains all jobs that are being executed at a certain moment.
- **self.idle_cores** defines a list with the number of cores that are not being used by a node. It is updated when scheduling a job or when it finishes its execution, and the goal of this global variable is to keep track of the unused cores to check if a job can be scheduled in a node at a certain time.

4.1.2 Common Functions

Every implemented algorithm derives from the *WorkloadManager* class already implemented in IRMaSim. *WorkloadManager* defines some methods shared by all scheduling algorithms. These are *on_job_submission*, *on_job_completion* and *schedule_next_job*.

- **on_job_submission** is called when a new set of jobs enters the platform. This function takes as a parameter the list of the entering jobs ordered by their ID. With this list, the function that plans the jobs is called, and then the scheduling of the set of entering jobs is added to the global scheduling. $self.scheduling \mathrel{+}= scheduling(J)$. Finally, the jobs are scheduled following *self.scheduling*.
- **on_job_completion** is called when a job or a set of jobs finish its execution, *on_job_completion* takes the list of finishing jobs and iterates over all lists that represent nodes in *self.scheduling*. When a job finds itself in the list, *self.scheduling* and *self.running_jobs* get updated by removing the job from both the lists. The variable *self.idle_cores* gets updated too by adding the number of cores, (which is the same as the number of tasks of the job) that have been unoccupied.

- **schedule_next_job**is a function called by *on_job_submission* and *on_job_completion*. It takes as a parameter the index of the node where the next job or jobs will be scheduled. Firstly, it iterates over all the jobs in *self.scheduling[n]*. If it is not running, checks if it enters the node by comparing its number of tasks with the idle cores of the node (*self.idle_cores[n]*). If the scheduling is possible, *schedule_next_job* updates *self.idle_cores* and *self.running_jobs* before allocating each task into a core.
- **scheduler**i a function that, although it is not defined by WorkloadManager, all algorithms described in Chapter 3 implement it. This function takes, at least, the list of entering jobs. It is called by *on_job_submission*, and schedules all entering jobs into the selected nodes. Then the scheduling of the set is returned in a variable called scheduling.

$$scheduling(j) = scheduling \neq self.scheduling$$

Starvation is a phenomenon that occurs when a job submitted to the platform is overtaken by another job that was submitted later, both of which are executed on the same node. This can result in the first job never being executed or in poor user-level efficiency. To prevent starvation, queued jobs can not be scheduled again and jobs are allocated following temporal and space guideline retrieved by the workload manager. To avoid starvation in IRMaSim, *on_job_submission*, *schedule_next_job* and *scheduler* functions only take the list of the lastly submitted jobs, without considering the ones that are in queued state.

4.2 Min-Min

Let's start by explaining the implementation of the *min-min* pseudocode 3.3.2 in IRMaSim. More precisely, the scheduler function.

The *scheduler* function of *min-min* takes two parameters. The first one is the list of entering jobs ordered by their *requested time* and the second one is the reference to the empty list *scheduling*. So that the first step of the pseudocode 3.3.2 is implemented in the *on_job_submission* function.

scheduler starts by making a copy of *self.cores_times* called *cores_times*. As well as the global variable, *cores_times* stores the *machine availability time* of all the cores grouped by their corresponding node. Then it appends a list for each existing node in *scheduling*, so that the two main structures are both initialized.

Afterwards, as indicated in the pseudocode 3.3.2, the list of jobs is iterated, however, instead of using the job itself, its index is used in order to access to the positions in *cores_times* and *scheduling* more easily.

For each job, a copy of *cores_times* is created, *cores_temporal_times* will be used to keep the results of the completion times of each task of the job in the cores of each node.

If $m(j_i) > m(n_k)$, the node is considered less favourable and is disregarded. This is achieved by assigning the maximum float value to the *machine availability time* of all cores in n_k.

On the other hand, if $m(j_i) \leq m(n_k)$, the estimation of the *completion time* is calculated by dividing the *requested_time* by the speed-up of the node and distributing it evenly among the cores associated with the minimum *completion time*. The number of cores allocated matches the number of tasks with the minimum *completion time*.

Once all the *machine availability times* are stored in *cores_temporal_times*, M_j is taken and compared to the maximum float value. If they are both the same, no node can allocate j_i, so *min-min* continues with j_{i+1}. Otherwise, the *machine availability time* of the cores of the node n (*cores_times*) is updated with the values of *cores_temporal_times* and j_i is appended to *scheduling* in the node n.

Finally, *cores_times* is returned by copy and, as explained before, *scheduling* by reference

4.3 Max-Min

As explained in Chapter 3, the *max-min* algorithm has the same structure as *min-min.* So that *scheduler* is the same for both algorithms. The difference is indeed when calling the function. It is called with a list of jobs ordered by the longest to the shortest.

4.4 Duplex

In *design*, duplex was defined as an algorithm that implements the best solution between *min-min* and *max-min.* That is why the scheduler code from the *min-min* and *max-min* algorithm is shared. Actually, in order to avoid repeated code, the three algorithms share the same code. This is possible thanks to IRMaSim, which allows executing itself with certain options. So the same workload manager can be selected with the option to execute *min-min max-min* or *duplex.*

These options are handled in the *on_job_submission* function. If the selected option is *min-min* or *duplex* the *scheduler* function is called with the list of jobs ordered by the *requested_time* as explained in *design*, and *min_min_scheduling*, that sets how all jobs from *j* should be scheduled. The returned value of *scheduler* is stored in *min_min_times*, which stores the final *machine availability times* of the nodes after the scheduling is done.

The same process is done with the *Max-min* or *duplex* option. However, the jobs are ordered by the inverse *requested_time* and the variables that store the scheduling and the *machine availability times* are *max_min_scheduling* and *max_min_times* respectively.

As indicated in the pseudocode 3.5.2, both maximum times in min_min_times and max_min_times are compared if the option is *duplex* and the option is changed into the one that implements the minimum time, so that the *duplex* option disappears. Finally, *self.scheduling* is updated with the values of *min_min_scheduling* or *max_min_scheduling* and the jobs of are scheduled.

4.5 A*

Finally, the implementation of *A**, will be explained, but firstly let's introduce a new global variable *self.init_cores_times.* This will substitute *self.cores_times* in order to store the relative *machine availability times* of nodes in relation to the previous set of jobs.

In *scheduler*, the first thing to do is initialize the tree. It is created by using the library *treelib.* This library provides an intuitive creation and manipulation of non-binary trees, allowing vertexes to have a non-binary number of children, as well as being implemented in python with minimum dependencies, and, the most important setting, it allows the visualization of the built tree.

Once the tree is created, the root vertex is generated. Treelib allows creating a vertex by setting a name and an identifier. Moreover, a field of data is provided in order to assign some value to each of the vertexes. To generate the structure and the mappings of all the nodes, various data needs to be stored in each vertex. So that, the data value is defined as a dictionary with several fields.

- **job**: The last job in the mapping representation of the current vertex. Notice is the whole job what is being stored and not its ID. This will lead into more temporal efficiency because the is no need to look for the job in the set *J*.
- **node**: The current node where the last job is being mapped. As well as it happens with the job, all the node is stored and not only its ID.

- **cores_times**: As defined in the previous algorithms, a data structure with the make span of the cores of the nodes. However, this make span is in relation to the previous set of jobs.
- **nodes**: A list with the identifier of the nodes where all the previous jobs in the partial mapping have been mapped.
- **fv**: A lower bound estimation on the ct(J, N) and the partial mapping that the vertex x represents.

As root represent an empty mapping, no *job* or *node* fields are in its data. Its *cores_times* field is computed by taking the maximum between 0 and the subtraction of the *machine availability time* for each core and the time elapsed between two sets of jobs enter the system. The vertex is added as indicated in the pseudocode 3.6.2, and the parent vertex is set as root.

In order to check if the mapping of *parent* contains all the possible schedulable jobs in J, the length of *nodes* plus the number of non schedulable jobs is compared to $m(J)$). If they are not the same, the next job is taken, and it is checked if it can be scheduled. This checking is made by comparing $m(j_i)$ with *self.max_cores*.

If the job is schedulable in any node $A*$ iterates over all nodes and after comparing if j_i is schedulable in n_k, if the tree has more than an established value of vertexes, pruning is performed. This number will be chosen by taking into account the results retrieved in a reasonable execution time. This pruning is done b selecting the maximum value of *layer_nodes*, a dictionary of leave vertexes where the keys are the identifiers and the values the $f(v)$ values. Then the vertex is located by its ID and extracted from *layer_nodes*, so that it will be never chosen to be the parent.

The next vertex is created with the current job and node, and its *cores_times* is initialized to its parent one. Afterwards, its *fv* is computed by calling a specific function *compute_fv* and the data is updated.

The selection of the next parent is similar to the pruning process, with little differences. Firstly, the value selected is not the maximum from the $f(v)$ in *layer_nodes* but the minimum. Finally, besides taking the vertex from *layer_nodes*, the *parent* variable is set to this node.

When a vertex with the mapping of all schedulable jobs is found, *scheduling* is updated by appending each schedulable job in the list with the index of each *nodes* values. Then *self.init_cores_times* and *self.prev_sim_time* are both updated too.

The function *compute_fv* calculates the $f(v)$ of the current vertex. This is done by computing the formulas in Chapter 3. In order to compute the values of $g(v)$ and *mmct* other two functions are used. *compute_gv* and *compute_mmct*. *compute_gv* takes the current job and node of the current mapping and uploads the values of *cores_times* in the data of the current vertex (they were initialized to the values of its parent). Then the $g(v)$ value is returned by taking the maximum value.

The *compute_mmct* function computes the *minimum completion time* for each job in each node. This is done in the same way as in *compute_gv* and the *scheduler* function in *min-min*, *max-min* and *duplex*. However, the *machine availability times* of the cores must not be updated, so a temporal list for each node that is restarted in every job iteration is used (*cores_temporal_times*). These M_i values are stored in a list, so the algorithm can return its maximum value.

5 Evaluation

In this chapter, a comparison between the performance of all the implemented algorithms will be performed. In addition, the performance of some already implemented algorithms such as *shortest_fastest*, *longest_fastest*, *OLB* and *random*, explained in Chapter 2, will be evaluated and compared to the implemented ones. In order to perform the simulations, all algorithms will be tested in a range of workloads and platforms, so that different scenarios can be evaluated.

5.1 Experimental Setup

To test how the algorithms perform, two sets of experiments will be conducted. In the first one, the proper functioning of the algorithms will be verified. For this purpose, an environment similar to the one used in [8] will be simulated, and the obtained results will be compared to those in the document. For the second set, a workload extracted from a real machine will be used, and the the behaviour of the algorithms will be evaluated on different platforms.

5.1.1 Heterogeneity Evaluation Environment

In [8] simulations are performed in an environment with a workload of 512 jobs composed by a unique task and a platform with 16 single-core nodes. In this environment, each node is capable of allocating any job. In [8] the *expected time to compute* for each job on each node is specified in a $J \times N$ matrix. These values are chosen through a random process. Even if these values make some sense, they are ultimately random. However, IRMaSim calculates the *completion time* of the jobs through with the number of instructions and the frequency of the processor. Thus, replicating these experiments is not entirely possible.

Due to the random nature of the setup used in [8], the platforms are not guaranteed to be consistent. Meaning that shorter jobs allocated on the same node may not necessarily take less time to execute than longer ones. This is shown in the *ETC* matrices in Table 2 and 3 in [8]. Fortunately, [8] does have some experiments that are consistent and these can be reproduced in IRMaSim.

A set of five different workloads have been generated. In order to imitate the original environment, each workload contains 512 single-task jobs. The requested time of the jobs in the set varies from 100 to 3000 seconds.

These workloads will be executed in two different platforms with different level of heterogeneity between their nodes. Table 2 shows the details of the low heterogeneity platform, and Table 3 shows the nodes of the high heterogeneity one.

Table 2: Platform With Low Heterogeneity

Low Heterogeneity Platform		
Number	**Cores**	**Clock Rate**
6	1	1-5 GHz
5	1	5.01-7.5 GHz
3	1	7.51-10 GHz
2	1	>10 GHz

Table 3: Platform With High Heterogeneity

High Heterogeneity Platform		
Number	**Cores**	**Clock Rate**
4	1	1-10 GHz
4	1	10.01-20 GHz
4	1	20-30 GHz
4	1	>30 GHz

5.1.2 Realistic Environment

KIT ForHLR II log is the workload used to perform realistic simulations. It is composed of 114355 jobs distributed in a time-lapse of a year and a half in a system located at the Karlsruhe Institute of Technology in Germany. After observing the workload, the period of time with most consistency of jobs submitted goes from the fourth month to the fifth one. So the workload has been reduced to consider only these count of jobs.

As explained in Chapter 4, the workload manager does not schedule jobs until they are submitted. And in order to prevent starvation, once they are scheduled, they can not be rescheduled. However, the workload *KIT ForHLR II log* only submits one job at a time, making the job of the workload manager trivial. Because of this, the submit time of the jobs has been rounded to 10^5. Jobs are now grouped by their submission day, allowing the algorithms to make more interesting scheduling decisions. These groups are composed of sets from 100 jobs to 300.

The number of jobs submitted to the platform in the selected time-lapse is close to 1500. So five different simulations with 500 jobs will be performed in two different platforms with 32 nodes. These are an approach to realistic platforms with nodes that could be found in a modern computer cluster. These platforms details are showed in Table 4 and Table 5.

Table 4: Distributed Heterogeneity Platform 0

High Heterogeneity Platform		
Number	**Cores**	**Clock Rate**
13	32	4.6 GHz
11	20	10.2 GHz
4	8	6.4 GHz
4	8	15.2 GHz

Table 5: Distributed Heterogeneity Platform 1

High Heterogeneity Platform		
Number	**Cores**	**Clock Rate**
4	32	2.6 GHz
4	20	5.2 GHz
4	8	3.4 GHz
4	8	7.2 GHz

5.2 Evaluated Time Metrics

The results of the experiments are evaluated through several metrics, which are defined as follows:

- **Total time** corresponds to the completion time of the workload $ct(J, N)$.

- **Average waiting time** refers to the average time jobs have being waiting to be executed. It is computed by taking the simulation time of the job in queued state and calculate its average.
- **Average slowdown** is the relation between the *execution time* of a job after being scheduled in a node and its *completion time*. It is computed by dividing these metrics. Then the average is calculated.
- **Average bounded slowdown** is computed by taking the slowdown of all jobs and exclude those under or over a certain value. This removes the negative effect of failed tasks that have short execution time and therefore very high slowdown. This metric is more stable than the slowdown.

5.3 Graphs Description and Results Analysis

For each different simulation, four bar graphs corresponding to each of the parameters previously described will be displayed. Algorithms are displayed horizontally, being the first four, the previously implemented ones. The four below correspond to the new implemented algorithms.

5.3.1 Low Heterogeneity Simulation

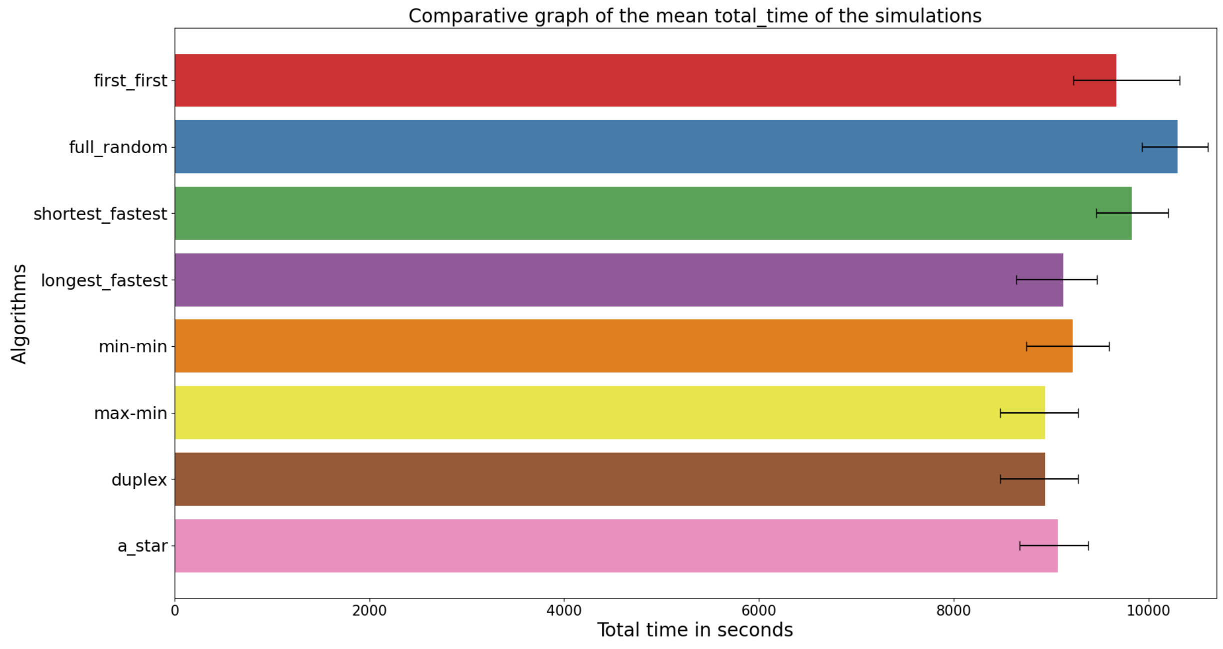


Figure 6: Total average time in a low heterogeneity platform

Figure 6 displays the average total time for the five workloads on the low heterogeneity platform for each algorithm. The total time of the simulation varies from the 8500 seconds to more than 10000. So there are noticeable differences between the performance. Simplest algorithms tend to perform worse, excepting the *longest_fastest* one, that gets better results than *min-min* and similar results to *A**. The best performing algorithm is *duplex*, tied with *max-min*. As jobs are only submitted in time 0, duplex only decides once which algorithm to use, choosing *min-min* and getting the same results. Both *longest_fastest* and *shortest_fastest* perform worse than *max-min* and *min-min* respectively. Finally, the performance of *A** is almost quite as good as the *duplex* and *max-min* algorithms, despite using a pruning factor of 128 vertices.

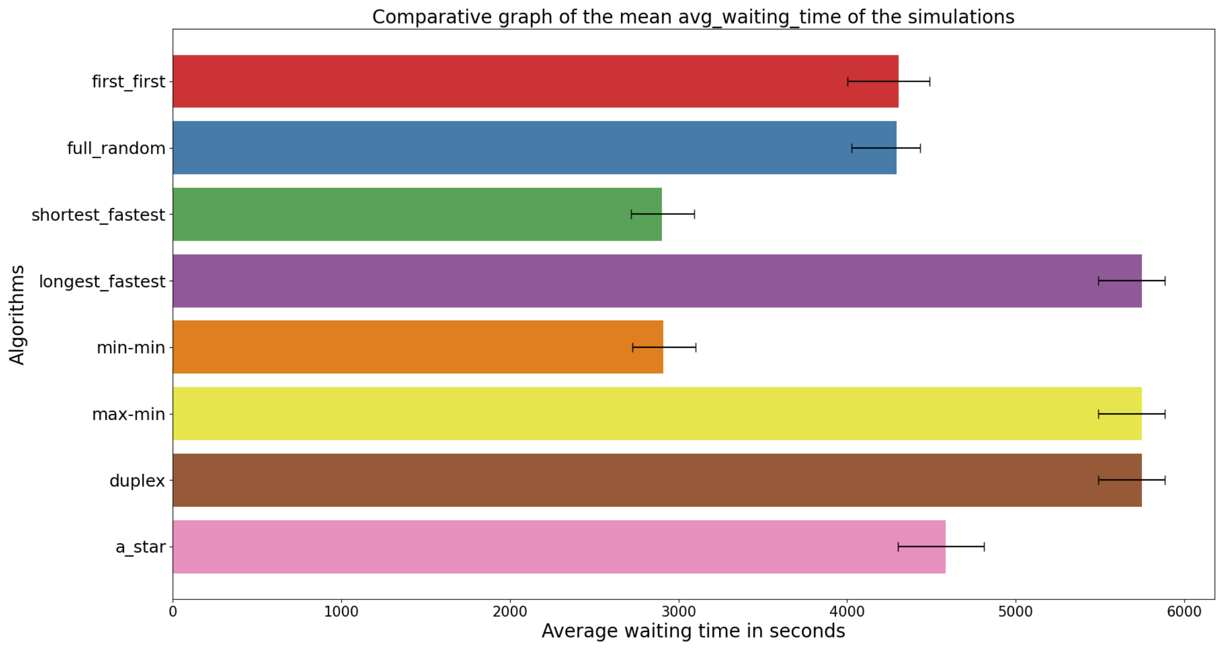


Figure 7: Mean waiting time in a low heterogeneity platform

The results of the average waiting time displayed in Figure 7 show a tendency for jobs in the *longest_fastest*, *max-min*, *duplex* and *A** algorithms to wait more. Apparently, the longest jobs are executed first, so the following ones must wait to start their execution. The opposite occurs in *shortest_fastest* and *min-min*, where jobs with shorter requested times are executed first, and longer jobs may start without waiting as long. Finally, *max-min* obtains better results than its counterpart *longest_fastest*.

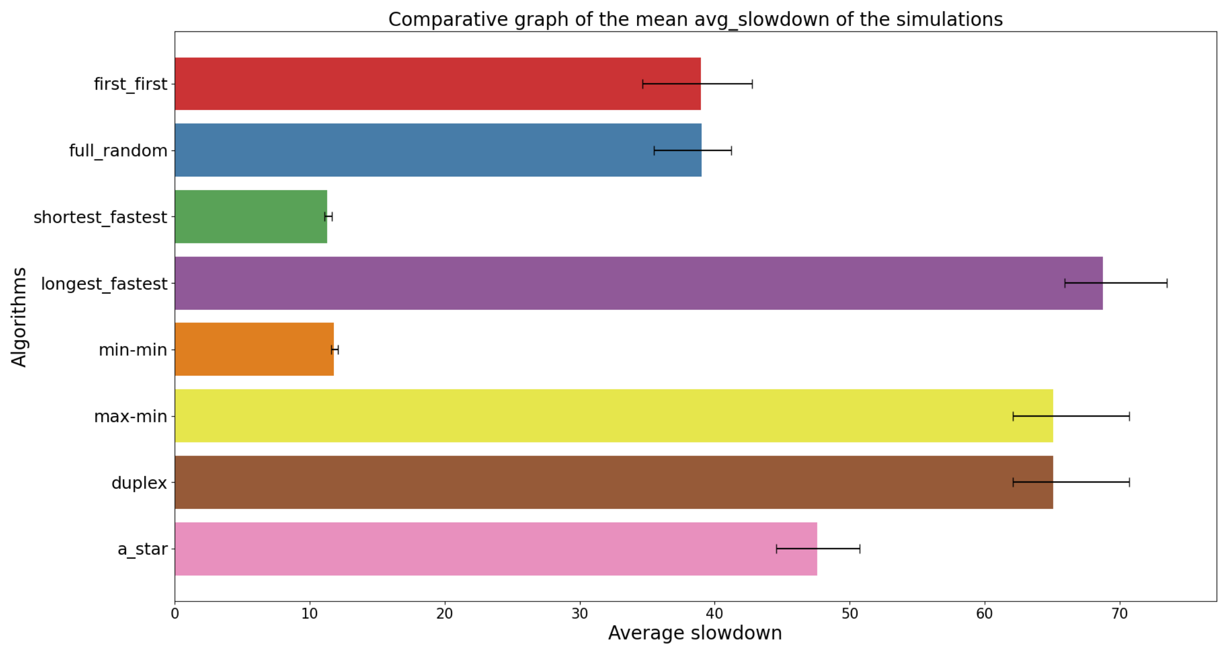


Figure 8: Average slowdown in a low heterogeneity platform

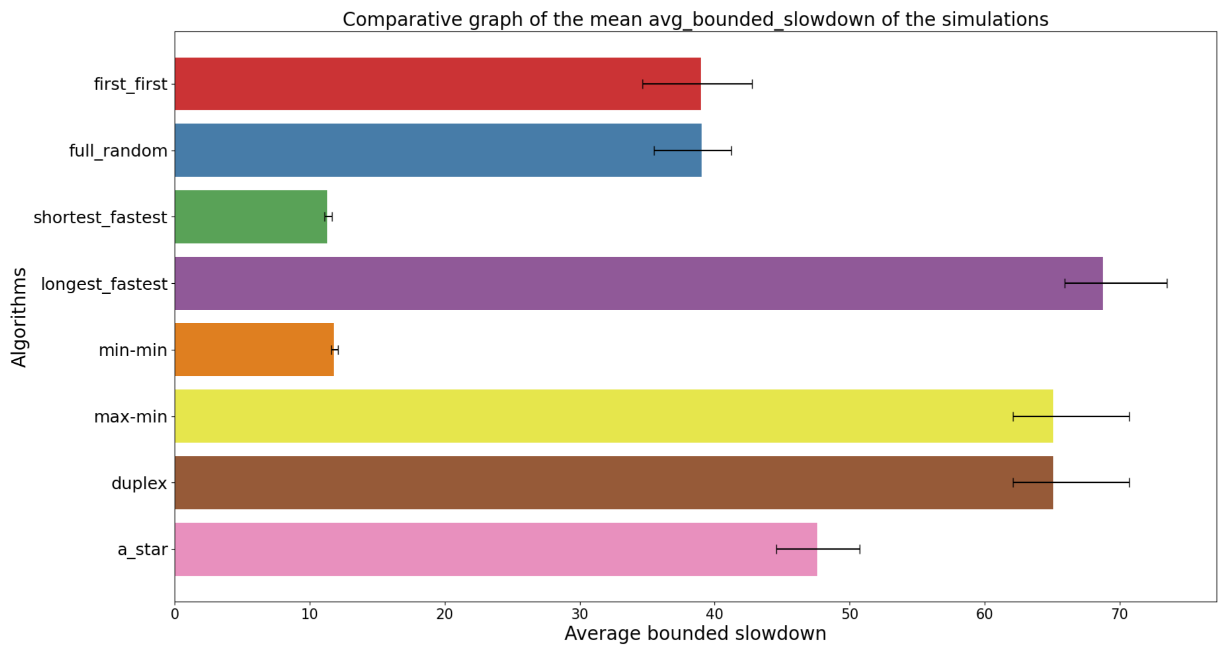


Figure 9: Average bounded slowdown in a low heterogeneity platform

Figure 8 and Figure 9 will be explained simultaneously, as although the results of bounded slowdown are better, tendencies of all algorithms remain the same. The best slowdown results are given by the *shortest_fastest* and *min-min* algorithms and being their standard derivation low too. On the other hand, *longest_fastest*, *max-min* and *duplex* get the highest values of slowdown, being six times the values of *shortest_fastest* and *min-min*. Finally, *first-first*, *full-random* and *A** retrieve slowdown values between 40 and 50.

When analysing all the results together, the conclusion is that there is not much correlation between the total time and the waiting time or slowdown is reached. Indeed, it is shown that algorithms with shorter waiting times achieve worse results (excluding the simplest heuristics). On the other hand, the newly implemented algorithms surpass the existing ones in all aspects in IRMaSim. *Min-min* matches *shortest_fastest* in terms of waiting time, slowdown, and bounded slowdown, while having a lower total time. However, *min-min*and *duplex* improve all the metrics compared to *longest_fastest*. Finally, *A** proves to be the most balanced, not achieving the best results in total time but obtaining good metrics in waiting time, slowdown, and bounded slowdown.

5.3.2 High Heterogeneity Simulation

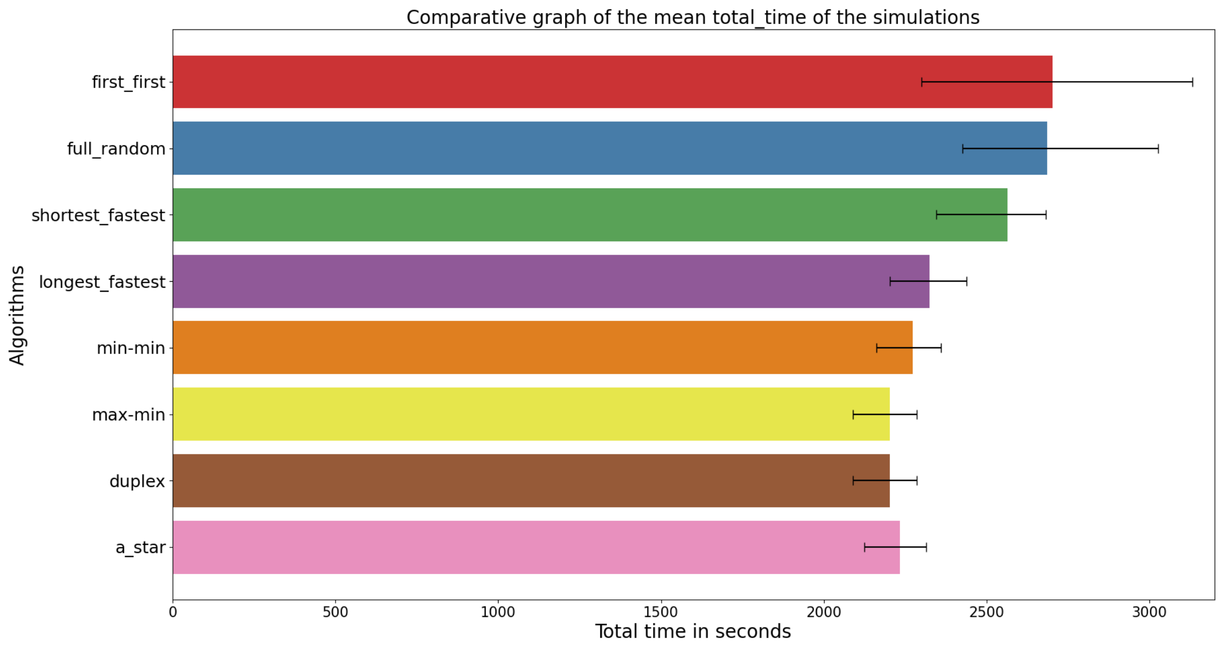


Figure 10: Total average time in a high heterogeneity platform

The total of the simulation shown in Figure 11 ranges from, 2100 to 3500 seconds. The simpler algorithms tend to exhibit poorer performance. Notably, the top-performing algorithm is *duplex*, tied with *max-min*. Both *longest_fastest* and *shortest_fastest* perform far worse compared to *min-min* and *max-min*, respectively. Additionally, *A** performs as the second-best algorithm, almost on par with *duplex* and *min-min*. Finally, the two simplest algorithms, *first-first* and *full random*, exhibit the worst average performance and a high deviation, resulting in simulations lasting over 3000 seconds.

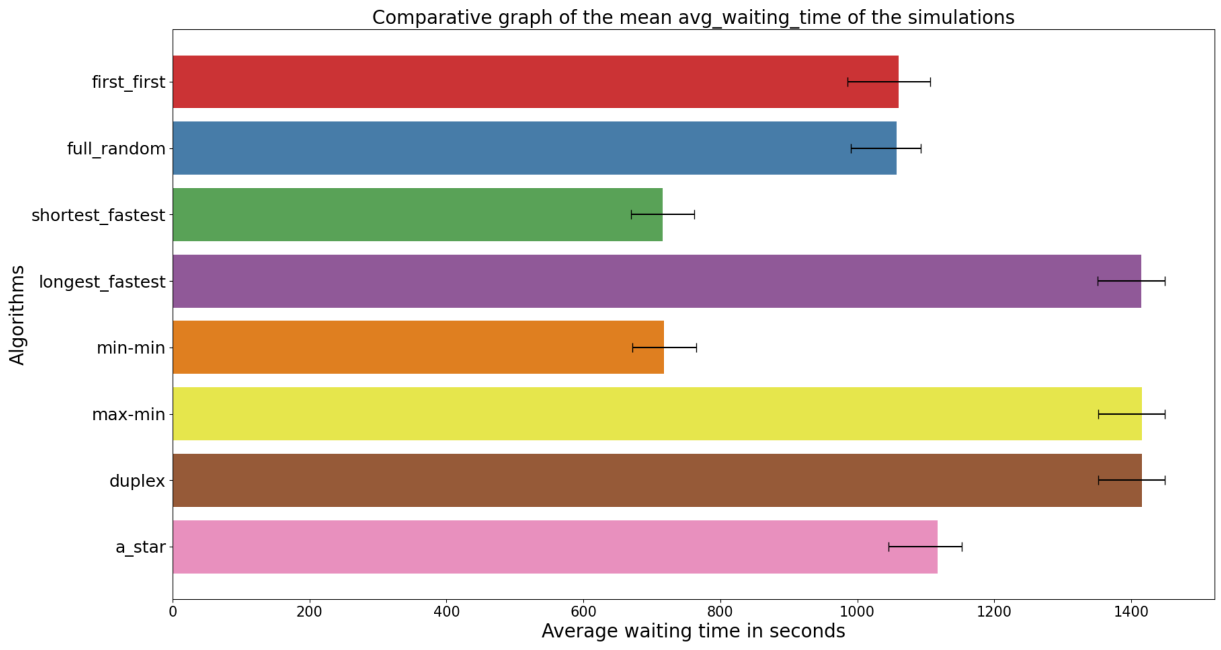


Figure 11: Mean waiting time in a high heterogeneity platform

The results of the average waiting time displayed in Figure 11 show poorer results *longest_fastest*, *max-min*, and *duplex*. This may happen for the reason explained in the low heterogeneity platform. Additionally, *max-min* performs better than its counterpart *longest_fastest*. Finally, *A**, *first-first*, and *full random* get similar results.

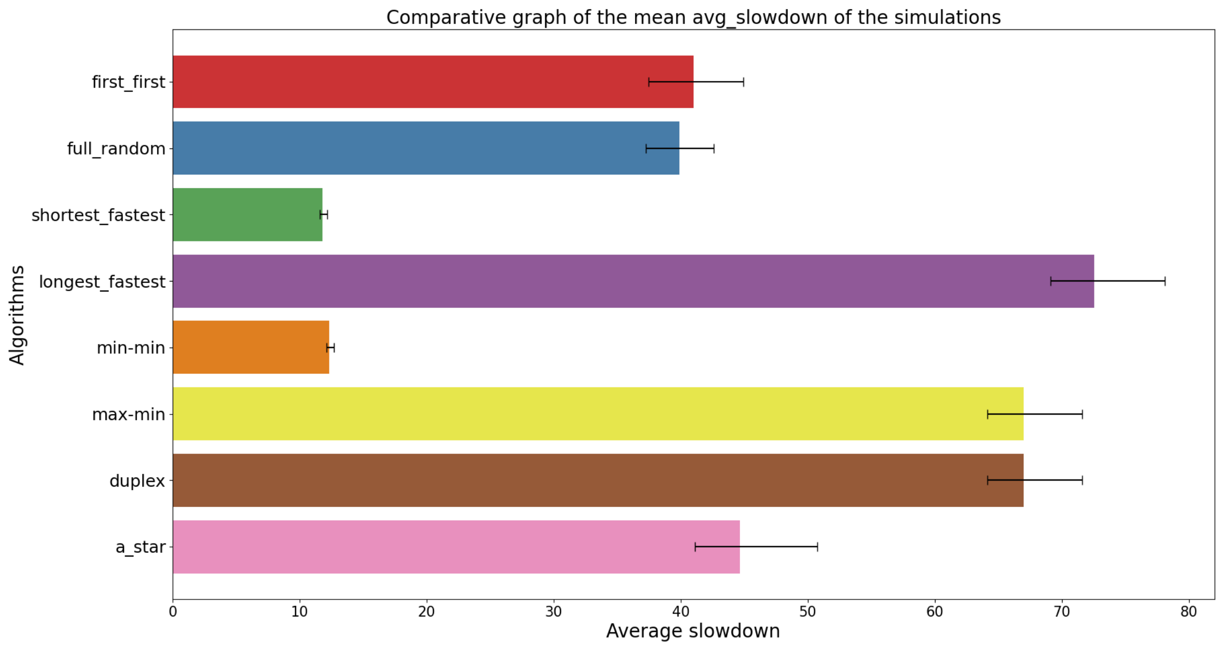


Figure 12: Average slowdown in a high heterogeneity platform

In Figure 12 the average of the slowdown of the jobs for all simulations is displayed.

Shortest_fastest is the algorithm with minimum slowdown, followed closely by *min-min*, with values between 10 and twenty. Then, *A**, *first-first* and *random* retrieve values from 40 to fifty units. Finally, *longest_fastest*, *max-min* and *duplex* jobs' average *completion time* has been over 65 times their *execution time*.

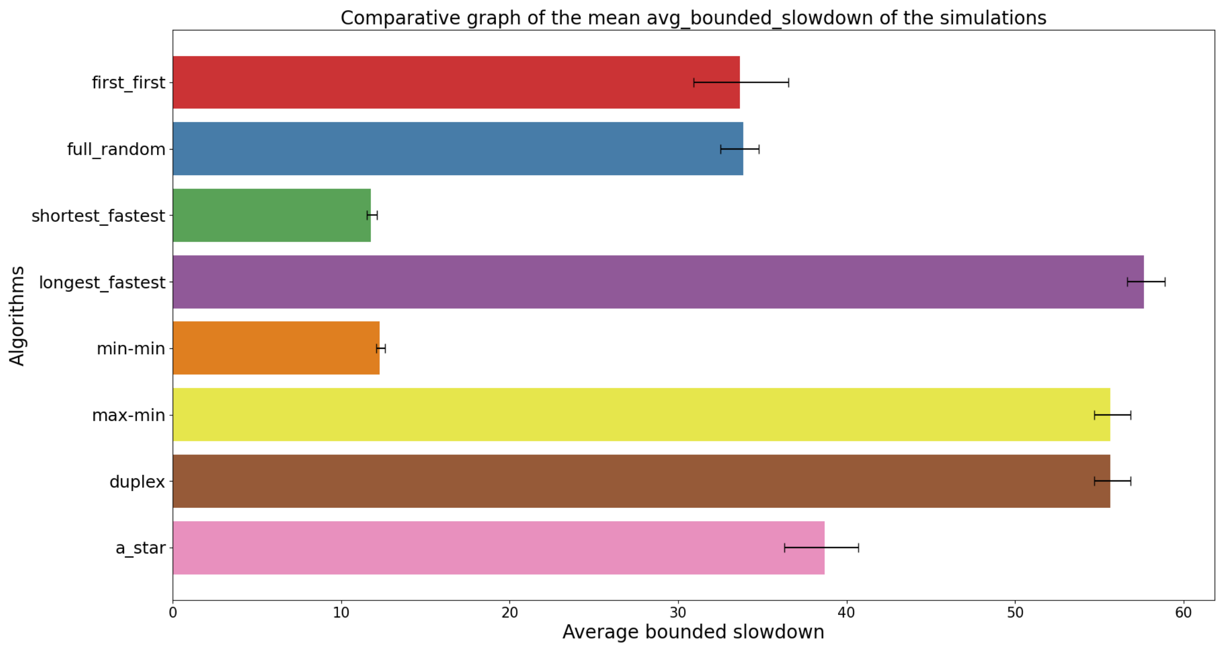


Figure 13: Average bounded slowdown in a high heterogeneity platform

Figure 12 presents the average bounded slowdown of jobs across all simulations. The *shortest_fastest* and *min-min* algorithms exhibit the lowest bounded slowdown. *A**, *first-first* and *random* yield bounded slowdown values ranging from 40 to 50 units. Lastly, *longest_fastest*, *max-min* and *duplex* surpass the 50 units of bounded slowdown.

Analysing all the metrics together, a trend is observed in the jobs of the *longest_fastest*, *max-min*, and *duplex* algorithms to wait longer before execution. Indeed, while the graphs of slowdown and bounded slowdown may not show the same numerical results, there is a strong correlation between the trends of each algorithm. This may be due to the normalization of jobs with execution time less than 10, resulting in lower values for bounded slowdown. Finally, it is observed that the average waiting time, slowdown, and bounded slowdown do not affect the total time, and as mentioned earlier, the newly implemented algorithms yield better results.

5.3.3 Comparison of Heterogeneity Results

Taking into account that the high heterogeneity platform is composed of faster nodes, it is observed in the graphs that the total time is lower. However, despite varying the heterogeneity, there are no significant changes in how the algorithms behave compared to the others. In both simulations, *max-min* and *duplex* exhibit better scheduling, followed by *A**. *Shortest_fastest*, *first-first*, and *full random* yield poorer results. The only noticeable difference is that in the low heterogeneity platform, the average of the *longest_fastest* algorithm is lower than that of *min-min*. In the high heterogeneity platform, this is reversed. For the rest of the graphs, the results are similar, with the same trends for all algorithms, although with better values in the high heterogeneity platform since, as mentioned, it is faster.

After comparing the results with those obtained in Figure 3 and Figure 4 of [8], similar performance among the algorithms can be observed. *OLB*, which is analogous to *first-first*, exhibits the poorest results. *A** does not achieve the best times but yields the worst outcomes, while *duplex* attains the best average total time by selecting the optimal option. However, the results for *max-min* and *min-min* are reversed. In [8], *min-min* achieves the best execution time, whereas in our simulations, it is *max-min* that exhibits better scheduling. Lastly, it is worth noting that changing the heterogeneity in the platforms does not lead to any significant shifts in the trends of the algorithms.

5.3.4 Real Workload With First Distributed Heterogeneity Platform

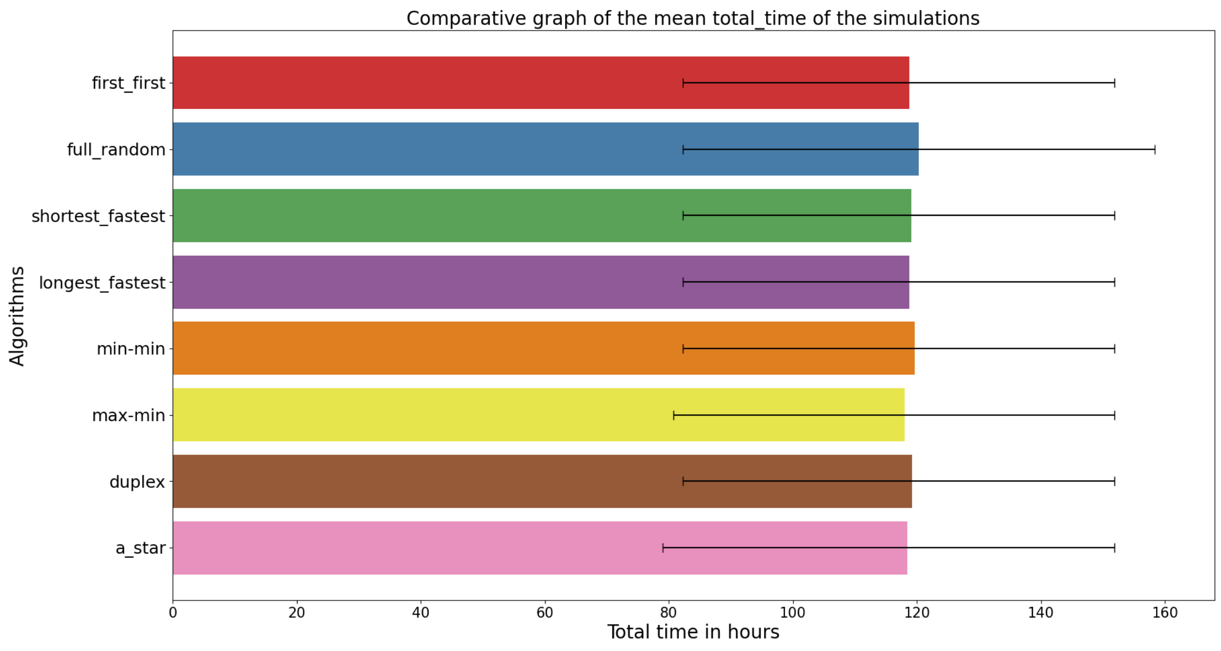


Figure 14: Average total time in the first platform with a real workload

In Figure 14 the average total time of the simulations is displayed, in hours. Small differences between the performance of the algorithms are shown, where all algorithms have an average total time close to 120 hours. However, there is some deviation between simulations with the same algorithm. *A** performs the best overall, with a minimum value of less than 80 hours. On the other hand, *full-random* performs the worst by almost reaching 160 hours in its longest simulation.

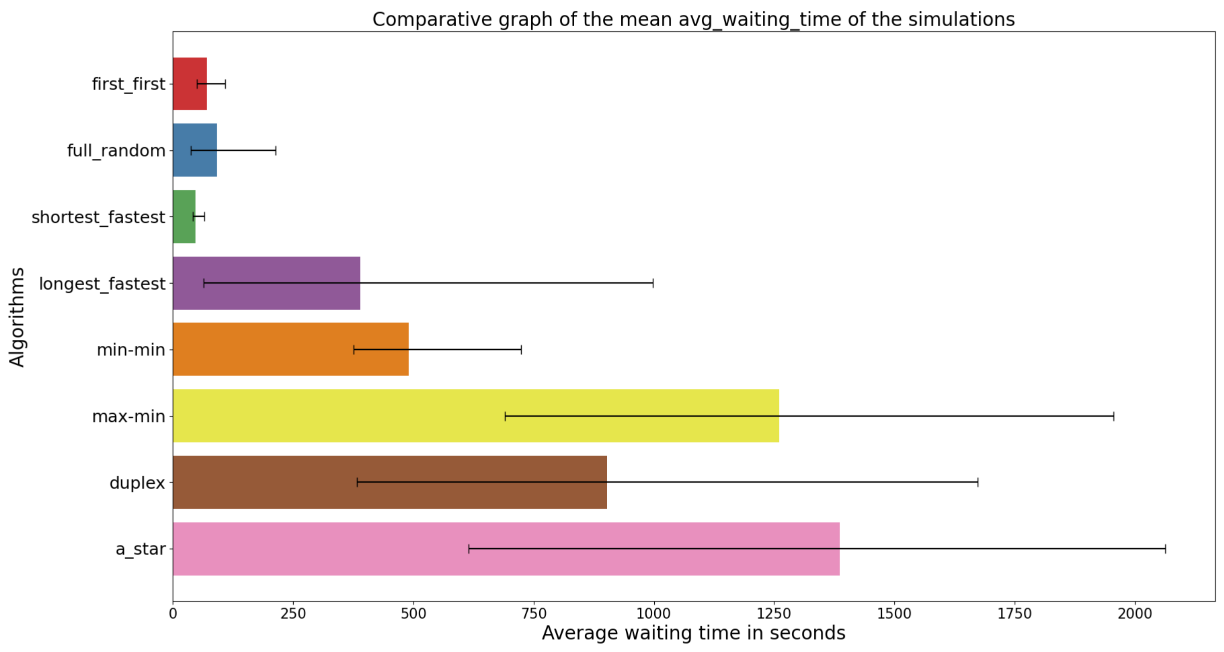


Figure 15: Average Waiting time in the first platform with a real workload

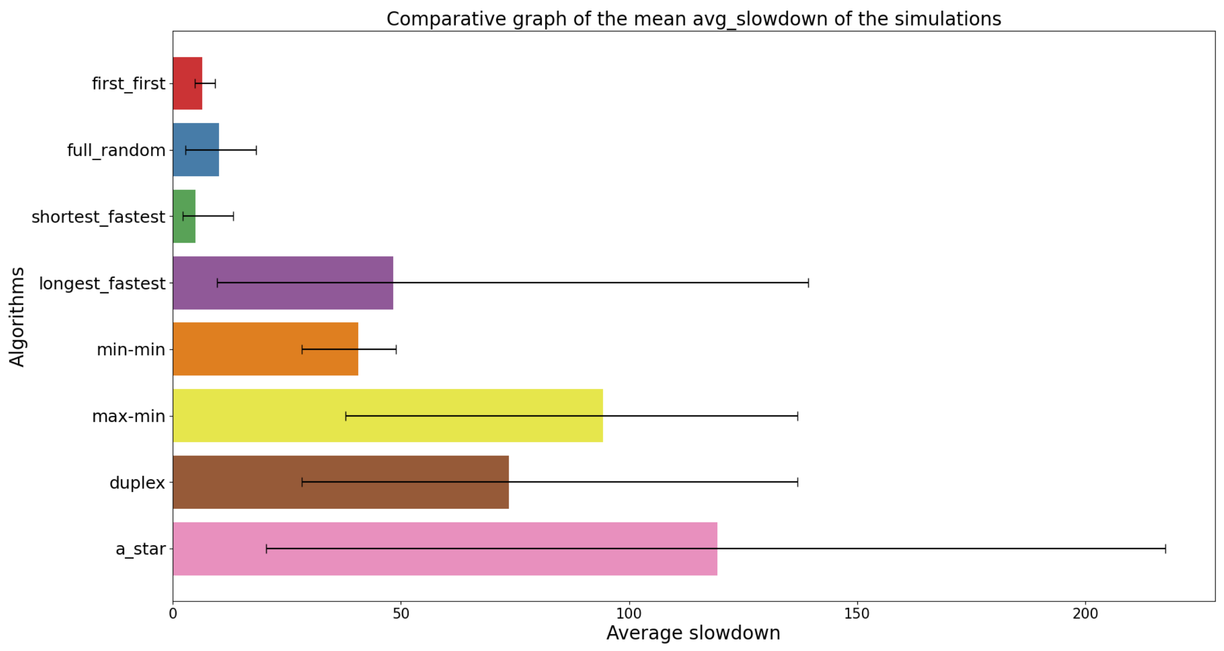


Figure 16: Average slowdown in the first platform with a real workload

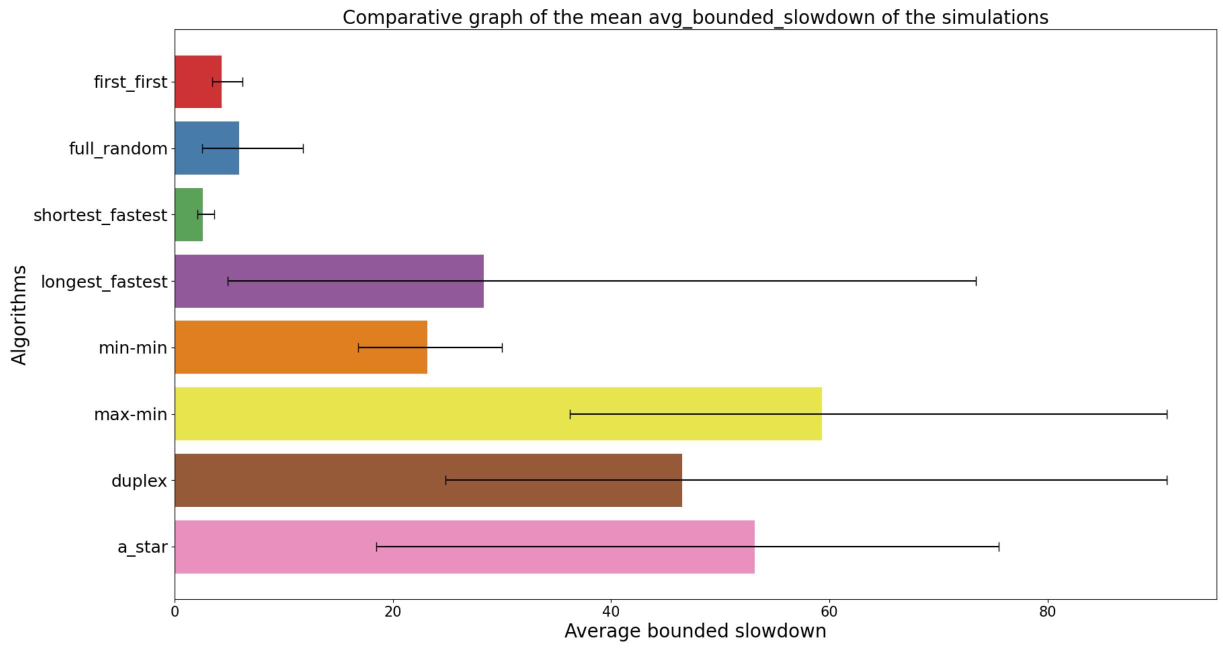


Figure 17: Average bounded slowdown in the first platform with a real workload

As similar results are obtained in Figures 15, 16 and 17, they will be compared together. The simplest algorithms are the ones where jobs have to wait the less time. There exists a relationship among the metrics for algorithms that prioritize shorter jobs, such as *shortest_fastest* and *min-min*, and those that prioritize longer jobs, such as *longest_fastest* and *max-min*. In both cases, the newly implemented algorithms yield inferior results compared to the existing heuristics in IRMaSim. Due to the varying arrival times of jobs, the *duplex* algorithm must make multiple decisions, resulting in intermediate outcomes between *max-min* and *min-min*. Ultimately, *A** algorithm exhibits the poorest performance in terms of average waiting time and slowdown. However, it demonstrates a significant improvement in bounded slowdown, which could be attributed to the presence of multiple jobs with very short execution times but high waiting times.

Upon analysing the collective results, it becomes apparent that while there may not be a significant distinction in total time, substantial variations exist in waiting time, slowdown, and bounded slowdown. These discrepancies can result in seemingly insignificant jobs, submitted at a particular time, taking an unexpectedly long duration to complete, which is suboptimal from the perspective of a user. Consequently, in these simulations, it appears that the *shortest_fastest* algorithm emerges as the optimal choice, as it not only achieves a favourable total time result but also boasts the lowest values among all algorithms for waiting time, slowdown, and bounded slowdown.

5.3.5 Real Workload With Second Distributed Heterogeneity Platform

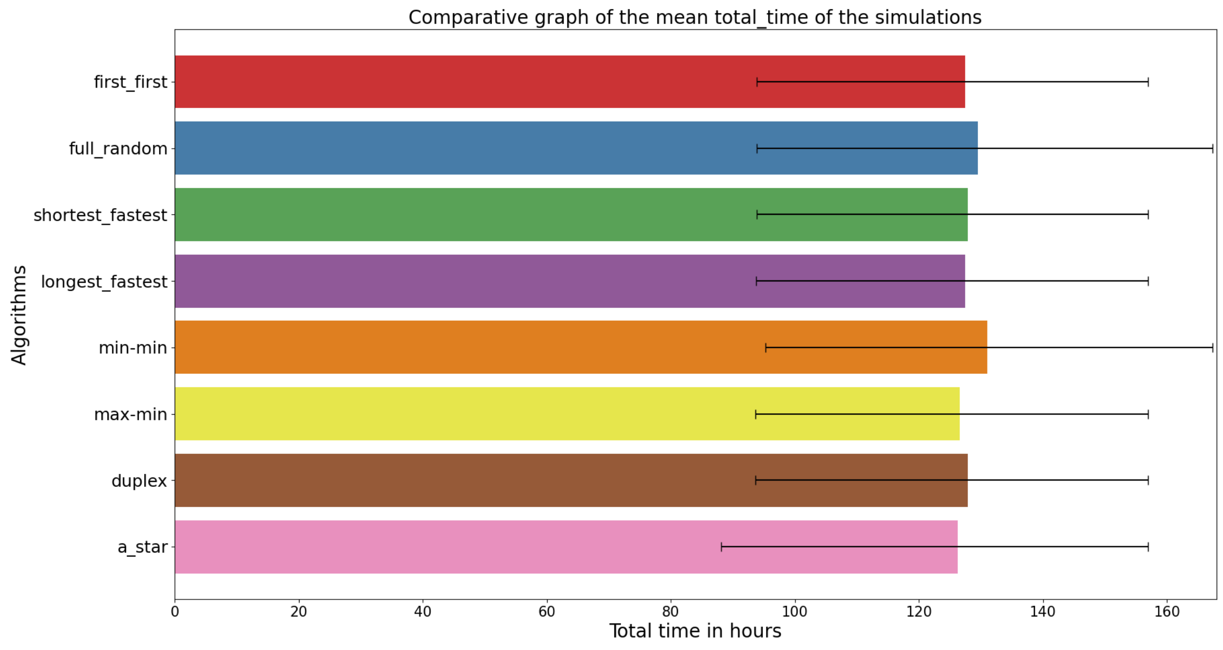


Figure 18: Average total time in the second platform with a real workload

Figure 18 shows differences between the performance of the algorithms in all simulations. *Min-min* obtains the worst result, with a mean value over 130 hours and a worse case over 160 hours. This approached is followed by the *full random* one. Another intriguing observation is that both the *min-min* and *max-min* algorithms outperform the duplex algorithm. Finally, the best perform is given by *A** with a simulation of under 90 hours.

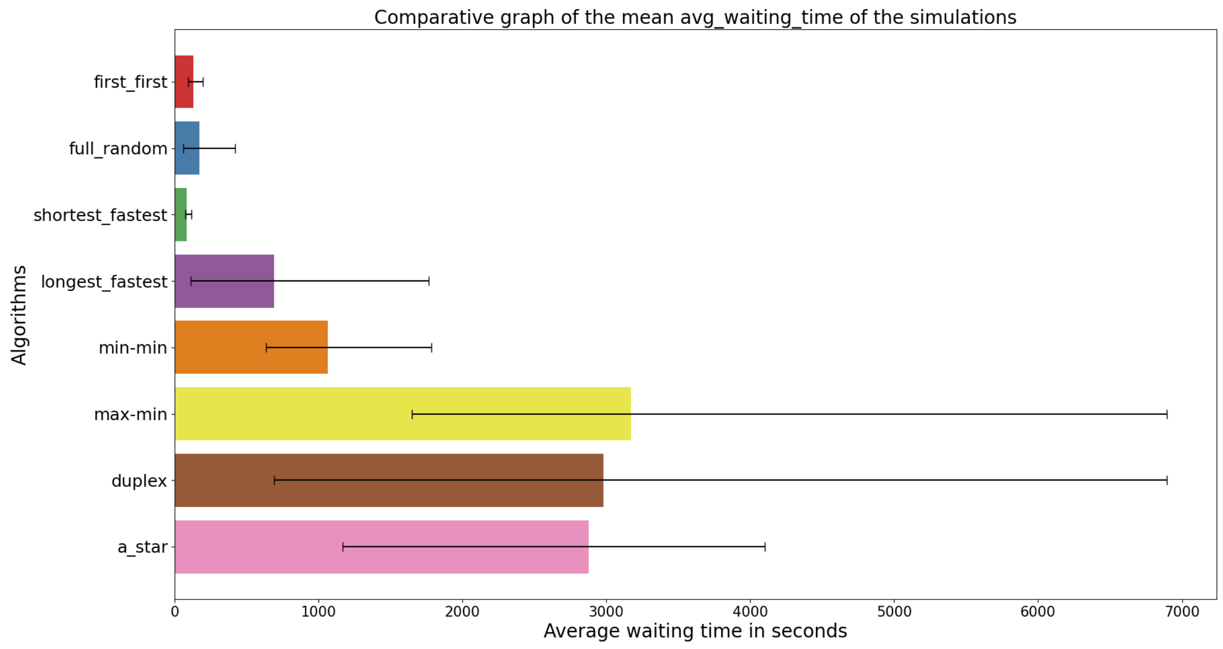


Figure 19: Average Waiting time in the second platform with a real workload

Figure 19 illustrates the waiting time for jobs in the simulation, revealing that the *first-first*, *full-random*, and *shortest_fastest* algorithms yield the least waiting time. The *min-min* and *longest_fastest* algorithms produce similar results, but the minimum value of *longest_fastest* is significantly lower than that of *min-min*, while the maximum values remain comparable. Therefore, on average, *longest_fastest* achieves better results. *Max-min* and *duplex* exhibit comparable performance, but the minimum value of *duplex* is on par with the minimum value of *min-min*. Finally, *A** outperforms *duplex* as its worst-case scenario is around 4000 seconds, whereas *duplex* approaches 7000 seconds.

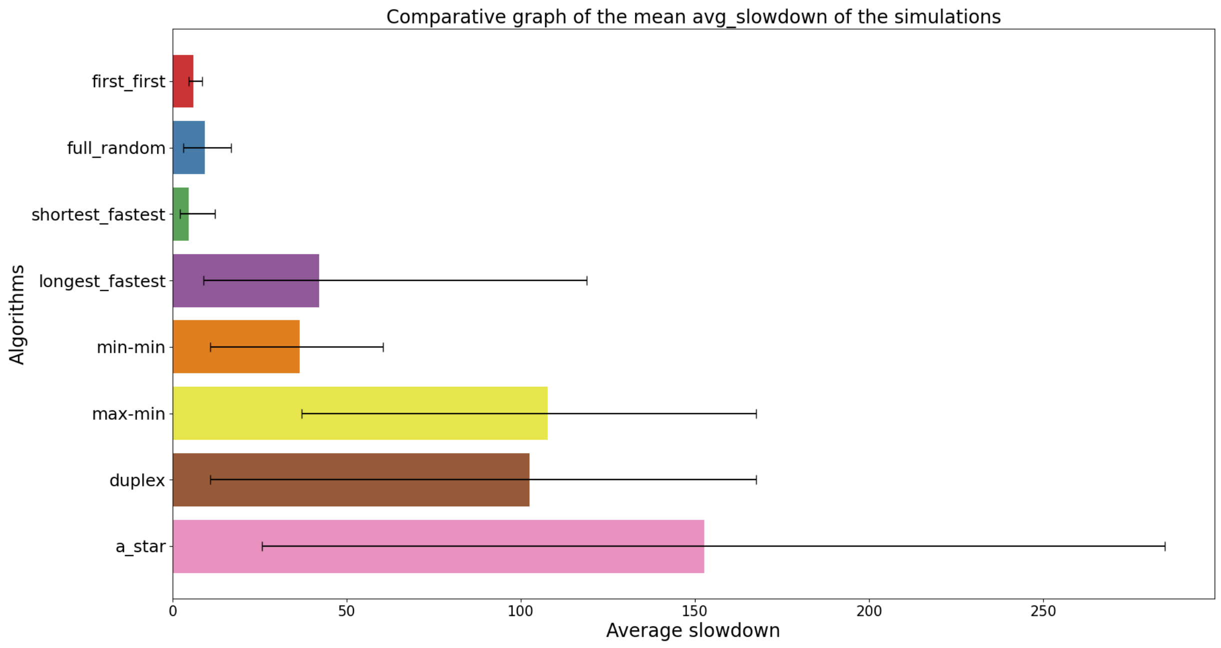


Figure 20: Average slowdown in the second platform with a real workload

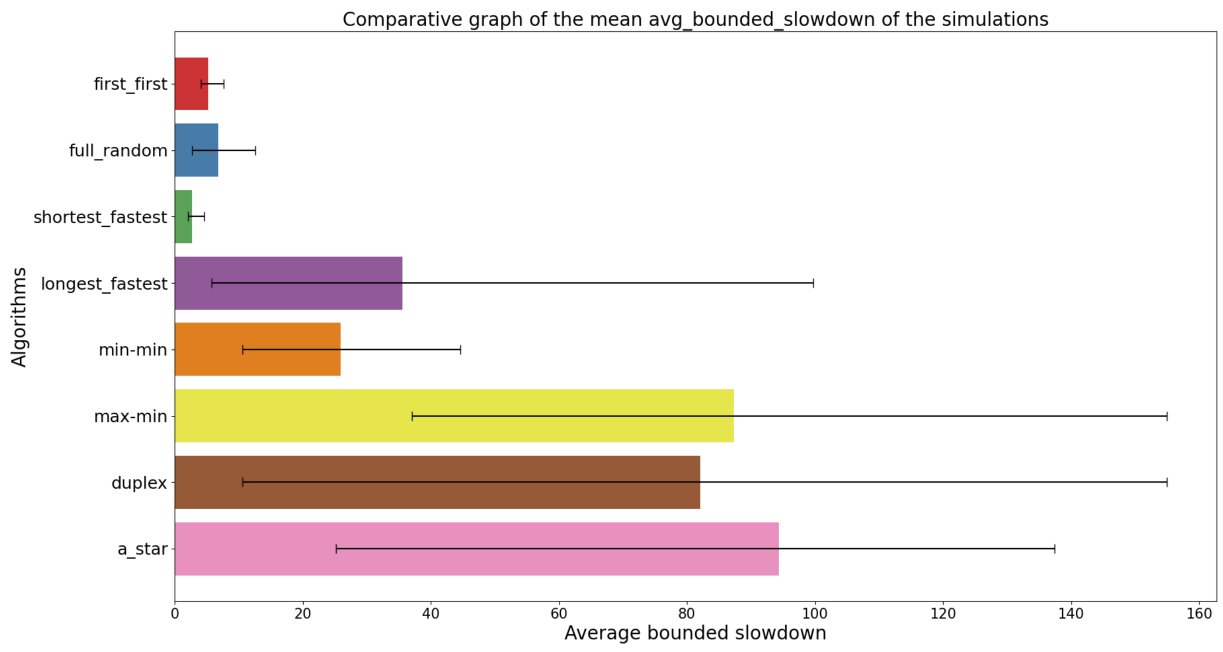


Figure 21: Average bounded slowdown in the second platform with a real workload

In 20 and 21 is shown how the *first-first*, *full-random* and *shortest_fastest* give the least waiting time for jobs in the simulation. *Min-min* and *longest_fastest* retrieve similar results, however the maximum value of *longest_fastest* is much higher than the *min-min* one. *Max-min* and *duplex* perform quite similar, but the minimum value of *duplex* is comparable to the minimum ones of *longest_fastest* and *min-min*. *A** gets high deviation results, which leads into the worst performance. However, the gap between *max-min* and *A** is reduced in the bounded slowdown. As the deviation is standardized, the simulation with the worst simulation slowdown, upgrades its value from over 250 to 140.

Upon examining all the obtained results in this platform, a notable trend is observed among the newly implemented algorithms, whereby they achieve better simulation times but experience an increase in jobs' slowdown. However, an exception to this trend is the *min-min* algorithm, which, due to its underperforming simulation, exhibits increased total time, slowdown, bounded slowdown, and waiting time. Moreover, the *duplex* algorithm yields results that fall between those of *max-min* and *min-min*. Finally, the *A** algorithm attains the best outcome in terms of total time, but it performs poorly in terms of slowdown and average slowdown.

5.3.6 Comparison of Real Platform Results

Both experiments have in common the similarity in results for the mean time. This may be caused by the time-lapse between the submitted time of the different groups of jobs. It can be possible that, for the platform 0, which is composed of faster nodes, the execution of all jobs in a certain submit time finishes before the new ones are submitted, so that the total time of the simulation is reduced to the last set of nodes. As nodes in platform 1 are slower, this more complicated for this scenario to take place. So the results from all algorithms differ more.

In terms of waiting time, there is a tendency of the simplest algorithms, being *first-first* and *random-random*, to perform worse. *max-min* tends to perform better than *longest-shortest* and, as there are multiple algorithm selections by duplex, its performance in between the *min-min* and *max-min* ones. Finally, A* performs satisfactory for both platforms.

When evaluating the remaining metrics, it is seen that the more complex is the algorithm, the more time jobs are tended to be waiting. For both platforms *first-first* and *full-random* get optimal values of slowdown, bounded slowdown, and waiting time, only surpassed by *shortest_fastest*, as it schedules first the shortest jobs. Duplex gets metrics between the *min-min* and *max-min* values. Finally, *A** tends to get the worst slowdown, for both platforms, in spite of retrieving the best average total time results.

5.4 Overall Result Discussion

From a global perspective, the results provided by the simulations evaluating platform heterogeneity exhibit greater variance in the performance of the algorithms in terms of the average total simulation time. With few exceptions, the algorithms described in this work consistently yield the best results in terms of total time. In fact, this difference is most pronounced in slower platforms. Regarding the other evaluated metrics, it has been observed that they are not necessarily proportional to the total time. *Shortest_fastest* consistently achieves the best values in terms of average waiting time, slowdown, and bounded slowdown, partly because it prioritizes jobs that are expected to take less time, as mentioned earlier. On the other hand, min-min, which follows the same philosophy, does not perform as well in real-world simulations. The opposite is true for max-min, which prioritizes longer jobs and significantly increases waiting time. Finally, the A* algorithm, despite employing a small pruning factor, attains very good temporal results while sacrificing a significant amount of job waiting time.

6 Conclusions

In this chapter, a brief summary of the work carried out during the development of the project will be provided, and an attempt will be made to address or analyse the final status of the objectives set at the beginning of this document. Furthermore, the main conclusions drawn by the author after completing this project will be discussed, as well as the major challenges encountered. Finally, a list of potential future work that could be interesting to engage to further enhance the project will be presented.

In Chapter 1 a set of objectives were defined. The first was to review some well known heuristic scheduling algorithms for heterogeneous clusters. A set of algorithms was selected after analysing several articles [4], [8], [5]. Then, their performance was compared by analysing the results given in [8], and the most efficient ones, *min-min*, *max-min*, *duplex* and *A** were selected.

After selecting the algorithms, their implementation in the IRMaSim simulator was initiated. This involved a software analysis and the implementation of these algorithms in the most efficient way. The result was the creation of two new workload managers. The first one that integrates the functionality of *min-min*, *max-min*, and *duplex* by choosing an option in its execution. The second one incorporates the functionality of the *A** algorithm.

Once the algorithms were implemented, some testing was performed. Three different results have been displayed throughout this document. A first set of simple simulations that provide an example of the behaviour of each algorithm. Then, in order to prove they work correctly, there has been provided a set of simulations in two different scenarios similar to the one in [8]. Both of them with similar results to the ones in the document. Finally, simulations with a real workload with two separate platforms have been executed, and their metrics have been evaluated. The results were optimal, as they present an overall upgrade in the completion time of the workload for the implemented algorithms.

For the author, though being harsh to implement both the algorithms and the real simulations, the completion of this project has been very rewarding, as it has provided a way to apply many of the theoretical knowledge acquired during the course of their studies, especially in the area of scheduling and algorithmic design.

Finally, a set of purposes of further investigation will be enumerated:

- An upgrade in the execution of the *A** algorithm. Although it retrieves good results, it takes a large time when the pruning factor increases over 128.
- Starvation prevention. As explained in Chapter 4, the implemented algorithms only schedule the newly submitted jobs. In Chapter 5, the real workload was edited in order to simulate a realistic but also a proper scenario to the simulations. To avoid this problem, the algorithms could take for the next scheduling the list of entering jobs plus the ones scheduled after some time ago. For instance, all jobs submitted last day.

References

[1] Esteban Stafford Adrian Herrera Mario Ibáñez and Jose Luis Bosque. "A Simulator for Intelligent Workload Managers in Heterogeneous Clusters". In: ().

[2] Ángel Arroyo et al. "Analysis of meteorological conditions in Spain by means of clustering techniques". In: *Journal of Applied Logic* 24 (2017). SI:SOCO 2015, pp. 76–89. ISSN: 1570-8683. DOI: `https://doi.org/10.1016/j.jal.2016.11.026`. URL: `https://www.sciencedirect.com/science/article/pii/S1570868316300830`.

[3] K.-W. Chow and B. Liu. "On mapping signal processing algorithms to a heterogeneous multiprocessor system". In: *[Proceedings] ICASSP 91: 1991 International Conference on Acoustics, Speech, and Signal Processing.* 1991, 1585–1588 vol.3. DOI: `10.1109/ICASSP.1991.150555`.

[4] R.F. Freund et al. "Scheduling resources in multi-user, heterogeneous, computing environments with SmartNet". In: *Proceedings Seventh Heterogeneous Computing Workshop (HCW'98).* 1998, pp. 184–199. DOI: `10.1109/HCW.1998.666558`.

[5] Oscar H. Ibarra and Chul. E. Kim. "Heuristic Algorithms for Scheduling Independent Tasks on Nonidentical Processors". In: *University of Minnesota, Minneapohs, Minnesot* ().

[6] Shubham Kamdar and Neha Kamdar. "big. LITTLE Architecture: Heterogeneous Multicore Processing". In: *International Journal of Computer Applications* 119 (June 2015), pp. 35–38. DOI: `10.5120/21034-3106`.

[7] Wael Khallouli and Jingwei Huang. "Cluster resource scheduling in cloud computing: literature review and research challenges". In: *The Journal of Supercomputing* 78 (Apr. 2022). DOI: `10.1007/s11227-021-04138-z`.

[8] Howard Jay Siegel Tracy D. Braun and Noah Beck. "A Comparison of Eleven Static Heuristics for Mapping a Class of Independent Tasks onto Heterogeneous Distributed Computing Systems". In: *Journal of Parallel and Distributed Computing* 61 (2001), pp. 810–837.

[9] Qu Wu et al. "Cluster expansion method and its application in computational materials science". In: *Computational Materials Science* 125 (2016), pp. 243–254. ISSN: 0927-0256. DOI: `https://doi.org/10.1016/j.commatsci.2016.08.034`. URL: `https://www.sciencedirect.com/science/article/pii/S0927025616304049`.