University of Minho
School of Engineering

Renato André Araújo Azevedo

Energy efficiency aware job scheduling for scalable data processing tools

october 2023

University of Minho
School of Engineering

Renato André Araújo Azevedo

Energy efficiency aware job scheduling for scalable data processing tools

Masters Dissertation
Master's in Informatics Engineering

Dissertation supervised by
Ricardo Manuel Pereira Vilaça

october 2023

Acknowledgements

I would like to thank Ricardo Vilaça, my supervisor, for assisting me through the entire process of making this dissertation, and the remaining members of the High-Assurance Software Laboratory (HASLab) for their help. Furthermore, I wish to thank INESC TEC for providing the necessary resources, facilities, and funding to make this research possible.

I also want to thank my family for supporting me through these five years, and my friends for always helping me when I needed it.

This work is financed by National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia, within project LA/P/0063/2020.

Statement of Integrity

University of Minho, Braga, october 2023

Renato André Araújo Azevedo

Abstract

Massive data processing tools for distributed environments such as Spark or Dask allow programmers to process massive amounts of data in data centers. A large portion of the operation costs of these infrastructures corresponds to the energy consumption resulting in performing these operations.

Current tools use simple algorithms for efficient scheduling of data processing jobs in distributed computing, relying on heuristics without considering the workload characteristics. Recent work explores efficient scheduling of data processing jobs in distributed computing, especially in heterogeneous environments, despite these infrastructures being typically homogeneous.

This dissertation makes an analysis of job executions in Spark and proposes EASAHUM a new algorithm for job scheduling in massive data processing tools with energy efficiency concerns using the conclusions drawn. The implementation and evaluation in a simulator using real and synthetic execution traces in Spark demonstrate that the algorithm can reduce energy consumption by up to 16% and reduce job execution time by up to 12.25% without significant impact on the scheduling time.

Keywords Spark, Scheduling, Energy Efficiency

Resumo

As ferramentas de processamento de dados massivos em ambientes distribuídos como o Spark ou Dask permitem aos programadores processar grandes quantidades de dados em centros de dados. Uma grande fatia dos custos de operação destas infraestruturas corresponde ao consumo energético resultante de processar estes dados.

As ferramentas atuais utilizam algoritmos simples para o agendamento eficiente de trabalhos de processamento de dados em computação distribuída, recorrendo a heurísticas sem ter em conta as características da carga de trabalho. Trabalho recente explora o agendamento eficiente de trabalhos de processamento de dados em computação distribuída, especialmente em ambientes heterogéneos, sendo que estas infraestruturas são tipicamente homogéneas.

Esta dissetação faz uma analise de execuções de trabalhos em Spark e propõem EASAHUM um novo algoritmo para o agendamento de trabalhos para ferramentas de processamento de dados massivos com preocupações de eficiência energética com as conclusões tiradas. A implementação num simulador e avaliação usando *traces* de execuções reais e sintéticas em Spark, demonstram que o algoritmo consegue reduzir o consumo energético em até 16%, além de conseguir reduzir o tempo de execução dos trabalhos em até 12.25%, sem grande impacto no tempo gasto no agendamento.

Palavras-chave Spark, Agendamento, Eficiência Energética

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

In recent years, the demand for computing power to process large quantities of data has increased and so has the energy required to do so. Since the data centers have to grant the client's needs, the processing power in these infrastructures necessary to process this large quantity of data will have to increase, which will consequently increase the energy needed to do so.

In Green500 [1], a list of the top supercomputers in the world in terms of energetic efficiency, we can see that the most powerful supercomputers already achieved 20 MW of power. According to studies, data centers are responsible for about 1-2% of global electricity consumption and will increase even more in the near future [4], and by 2030, it is expected the ICT to be responsible for 20.9% of the global energy demand [18], where data centers are one of the main contributors with 36% of the total ICT energy consumption [24]. In addition to that, the increase in energy consumption will lead to more cooling needs, which will further increase the data center's budget. With such a large amount of energy in use, even a small optimization will reflect a huge saving.

1.2 Problem description

Apache Spark [41] is currently widely used in data centers to process large quantities of data, and by default it does not provide any options for energy saving [28]. By default, it has two scheduling algorithms: FIFO and FAIR [2], and both algorithms are very simple and do not have any considerations regarding energy consumption when scheduling the jobs, so an energy-aware scheduling algorithm may make it possible to reduce the energy consumption of applications running in this environment. With this, we will be able to reduce not only the operational costs but also carbon emissions.

In recent years, many scheduling algorithms have been proposed to reduce energy consumption in heterogeneous clusters by taking advantage of the difference in machines to meet the needs of the different jobs. However, data centers are typically homogeneous so they cannot use the full potential of these algorithms.

1.3 Objectives

The objective of this dissertation is to create an energy-aware scheduler capable of efficiently scheduling jobs on a big data framework and reducing the energy consumption on a cluster, without compromising the performance. We expect that the correct placement of the tasks within the cluster can reduce energy consumption, in order to reduce carbon emissions in these facilities and also reduce the budget spent on energy.

To achieve this made an extensive analysis of various workloads, from different use cases from a test framework and identified the main characteristics that lead to energy consumption in these workloads, in order to implement the scheduler algorithm.

After developing and implementing the scheduling algorithm, we will analyze the energy reduction and performance impacts on a test environment, comparing the new scheduler against the default schedulers available on Spark, such as FIFO and FAIR, or developed in other papers. To register the energy consumption with precise measurements, we will use specialized software that registers the power consumption every second during all the tests.

1.4 Structure

This dissertation is organized as follows. In chapter 2 we show relevant concepts for the understanding of this dissertation, as well as review some of the relevant work in this area, especially scheduling algorithms that focus on reducing power consumption. In chapter 3 we run a set of Spark jobs from a benchmark tool and compare the default scheduling algorithms of Spark (FIFO and FAIR) in terms of energy consumption and execution time, and then we rerun the same workloads with the FIFO algorithm to analyze how does the frequency and the number of cores affect the power consumption. In chapter 4 we propose the algorithm and explain how it was designed. In chapter 5 we present the simulator used to implement and test the proposed algorithm, as well as the energy model used to measure the energy consumption. In chapter 6 we present and analyze the results of running the same workloads used in chapter 3 in the simulator, and compare the results with the ones obtained with an energy-aware version of FIFO. Finally, chapter 7 concludes this dissertation and presents what can be done for future work.

1.5 Results

After comparing the results obtained with the FIFO and FAIR algorithms, we concluded that there is no significant difference between them for the tests used. While running tests in Spark, we also confirmed that the power consumption can be reduced by reducing the frequency of the processor at the cost of increasing the execution time. We also confirmed that energy consumption can be reduced by using more cores, especially when the processor is not running at high frequencies.

With the information from the previous tests, we designed EASAHUM (Energy Aware Scheduling Algorithm Host Underusage Minimization), an energy-aware algorithm that aims to reduce the energy consumption in homogeneous clusters by reducing the host underutilization, with a focus on Spark workloads. After testing our proposed algorithm, we concluded that it is able to reduce power consumption by up to 16% and even reduce the execution time by up to 12.25%, where the best results happen when the workload is composed of multiple tasks with different execution times, and also show better results for Spark workloads.

Chapter 2

State of the art

2.1 Background

2.1.1 Energy efficiency

During the last few years, there have been various studies around energy efficiency techniques in infrastructures such as data centers due to their high energy consumption, intending to reduce carbon footprint. In a study made by Anderson et al. [4], they develop mechanisms to reduce carbon emission that include: reducing the software bloat, interchanging computational and memory resources, and scheduling policies. In [10] they also point out techniques involving dynamic voltage and frequency control, and power capping. In [11] multiple experiments are performed on deep network training and the power capping was adjusted for multiple values during the tests. They concluded that power capping the GPU has significant power savings without compromising the performance, where 200 W was the optimal value for their tests, resulting in a similar performance to 250 W but with higher energy savings.

One important aspect while analyzing energy efficiency is how we will measure energy consumption, to make sure we are effectively achieving energy efficiency and not having measurement errors. There are three ways to do this, which are:

- Directly measuring the power consumption using specialized hardware. The use of this technique is only possible if the infrastructure has this specific hardware installed in their machines.
- Using software to measure periodical the consumption of each component. This will only be possible if the components support this feature. However, nowadays, tools like Powerjoular [26] support every machine with a RAPL supported Intel or AMD CPU, and optionally an Nvidia GPU, in a GNU/Linux environment.
- Creating a theoretical energy model and making an approximation to the energy consumption.

The choice of each technique will affect the measurement errors, where the first one has the most approximate value, and the last one will produce results with high approximation errors. Since Intel and AMD are widely used alongside GNU/Linux, in this type of environment, using software libraries to measure the power consumption will be the best option, since we will not require an infrastructure with specialized hardware, and the measurements will have a low approximation error. The usage of an energy model also requires us to model many parameters and has no advantages over the usage of a software library.

The power consumption of each machine can be mathematically represented as the sum of the static power P_{static} and the dynamic power $P_{dynamic}$, as shown in equation 2.1 [29, 38].

$$P = P_{static} + P_{dynamic} \tag{2.1}$$

While the static power P_{static} represents the amount of power consumed by the machine in an idle state, which we can also call leakage, the dynamic power $P_{dynamic}$ represents the amount of power consumed due to circuit activity in the machine [36].

The dynamic power can be calculated using the equation 2.2 [12, 30].

$$P_{dynamic} = A \times C \times v^2 \times f \tag{2.2}$$

where:

A is a constant that represents the switch activity

C is the capacitance

v is the supply voltage

f is the frequency of the processor

2.1.2 Dynamic Voltage and Frequency Scaling (DVFS)

As mentioned before, controlling the voltage and/or the frequency of the CPU is a technique used to reduce the power consumption in the CPU. This technique is known as Dynamic Voltage and Frequency Scaling (DVFS). It allows the CPU to work under different frequencies and/or voltage, resulting in lower power consumption, since power varies quadratically with the voltage [39], as we saw previously in equation 2.2. Decreasing the frequency also has a great impact on power consumption not only due to its direct impact but also because the processor will require a lower voltage to operate at a lower frequency [10, 19].

Although this technique can reduce power consumption, it may greatly increase the execution time and result in lower energy reduction than expected. If we lower the frequency of the processor, it will also reduce the number of instructions it can execute per second, which makes the execution of the tasks slower. As we can see in the equation 2.3, the time T has a direct impact on the energy E. So, even if P decreases, T will in turn increase leading to lower energy reduction than expected.

$$E = P \times T \tag{2.3}$$

where:

E is the energy

P is the power

T is the time interval

2.1.3 Schedulers

A scheduler is a piece of software that whenever a task is submitted to the cluster, decides which resources that job will use to execute. These resources can be a machine or even a process that will run the task, and the scheduler must know the state of the resources in order to work.

So, the scheduler has the responsibility to assign any given tasks to a worker, which means that given a set of tasks $T = \{T_0, T_1, ..., T_n\}$, and a set of workers $W = \{W_0, W_1, ..., W_m\}$, the scheduler must assign each task T_i to a worker W_j, where the result of this operation is a set $S = \{S_0, S_1, ..., S_r\}$, where each S_k is a pair (T_i, W_j) representing which worker W_j the task T_i is assigned to.

To assign each task to each worker, the scheduler must use a scheduling algorithm. Since the scheduling problem is NP-Complete [35], many heuristics have been created to solve this problem. Two simple and famous scheduling algorithms used in many areas are FIFO [42] and FAIR [17].

The first algorithm, FIFO, uses the first-in-first-out method. This means that whenever a task arrives at the scheduler, it will be placed in a queue and wait until it is the first in the queue to execute. So, when a task executes, it will try to use all available workers. Figure 1 shows an example of resource allocation using the FIFO algorithm.

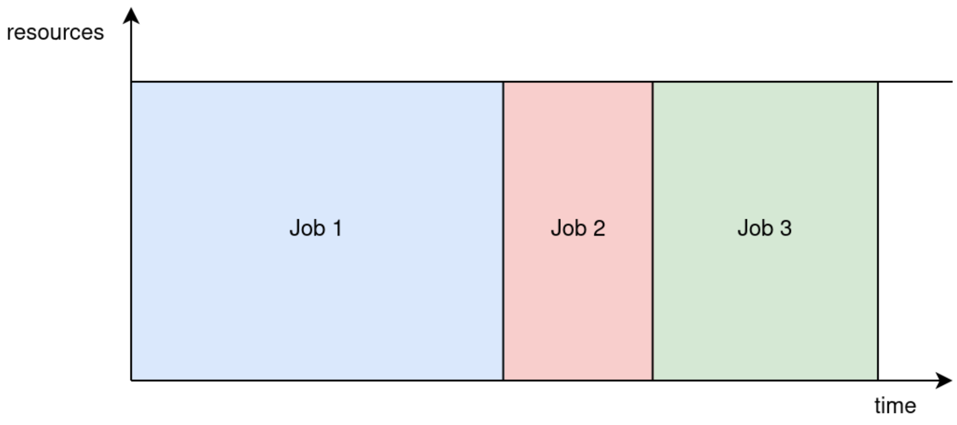


Figure 1: Example of resources utilization with FIFO

As we can see in 1, each task will only free the resources when it finishes. This can cause idle workers to wait for other workers who have not finished yet. Another problem is that if a priority job arrives at the scheduler after many fewer priority jobs, our newly arrived priority job will have to wait for all fewer priority jobs, which will bring problems regarding deadline constraints [25].

The FIFO algorithm has some variants, and one of them that is relevant for this work is the energy-aware version E-FIFO [23]. This version of FIFO can turn off hosts that are not in use in order to save energy. The previous example 1 is a very simple and realistic example because the job will be concluded at the same time in all resources. In reality, some resources will finish their work earlier than others, and if those machines are kept in, they will consume energy. This way, E-FIFO can save some energy without any changes to the rest of the algorithm.

For the second algorithm, FAIR, the scheduler will try to assign the jobs in a way that the resources are equally shared between them. This means that if only one task is running in the cluster, it will hold all the resources, but if two tasks are running, then each one of them will roughly share half of the resources, and so on.

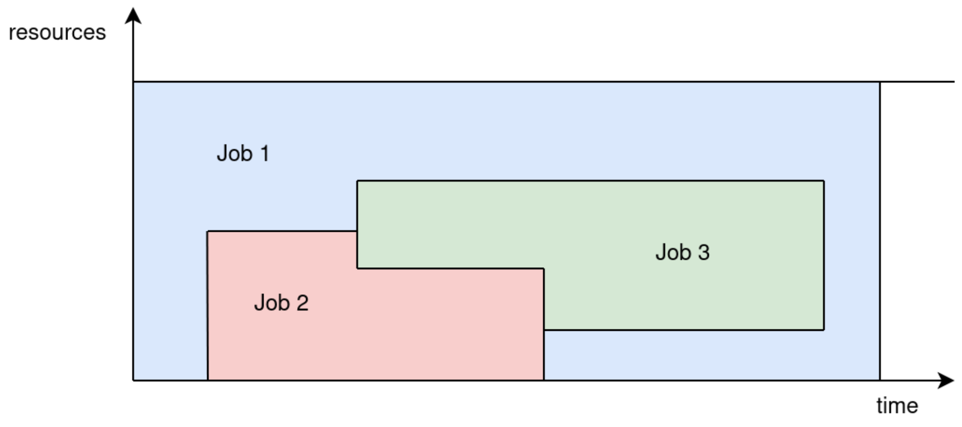


Figure 2: Example of resources utilization with FAIR

As we can see in 2 when Job 1 starts running, it holds all the resources, but when we submit Job 2, Job 1 starts using only half of the resources so that Job 2 can start running right after. Then, when Job 3 is submitted, Job 1 and Job 2 will hold a third of the resources so that Job 3 can start running. This way, each job will hold roughly the same share of the resources, and when it finishes, the remaining jobs will hold the freed resources. Thus, even if a long-running job is running (in this case Job 1), other smaller jobs (Job 2 and Job 3) will be able to execute.

Another advantage of the fair scheduler is that we are able to define pools and reserve resources for each pool, for example, we can create a priority pool that holds half of the resources and whenever a job with high priority arrives at the scheduler, we can place that job in the priority pool in order to reduce its execution time.

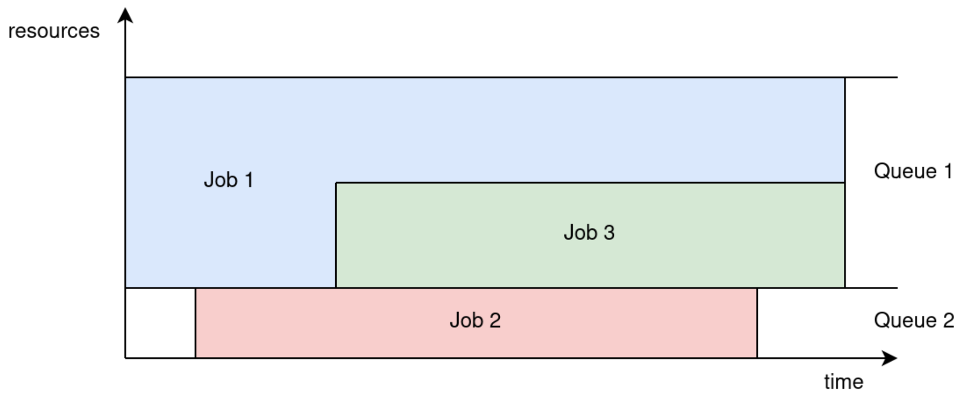


Figure 3: Example of resources utilization with FAIR using a priority queue

In the example shown in 3, there are two queues: Queue 1 and Queue 2, where Queue 2 is treated as a priority queue. We are considering Job 1 and Job 3 regular jobs and Job 2 a priority job. When we submit Job 1 to Queue 1, it will no longer use all available in the cluster, since it will only have access to 75% of the resources. Only when we submit Job 2, that 25% of the resources are finally used. Note that in this example, the execution time of Job 2 will be lower compared to the previous example where we didn't use the priority queue, but if we submit at least 2 more tasks at the same time, we would only have access to 20% of the resources in the latter case, while we still have 25% in this one

It is relevant to note that none of these algorithms takes into consideration the characteristics of the jobs or the workers. This happens because these are simple algorithms created for general usage, however, they will not provide good results in a specific scenario. In an effort to optimize the scheduler and make them suitable for a specific scenario, researchers tend to create algorithms that take into consideration the characteristics of both the jobs and the workers, as we will see later. In our case, we want the scheduler to assign the tasks to the works in a way that the energy consumption is lower without violating job deadlines.

Other famous algorithms are Heterogeneous Earlist Finish Time (HEFT) and Critical Path on a Processor (CPOP) [34]. There are also other algorithms like First Fit (FF) [16], Best Fit Decreasing (BFD) [13], and Greedy Iterative Optimization (GIO) [16]. These algorithms are also resumed in [32].

In the First Fit algorithm (FF), the scheduler will iterate through the list of pending tasks and then find a worker node that meets the requirements to execute the task. This way, the pseudo-code can be written as in 2.1

```
for task in pendingTasks:
    for worker in availableWorkers:
        if worker meets the requirements:
            place the task in the worker
            break
```

Listing 2.1: FF pseudo-code

Best Fit Decreasing (BFD) works similarly to FF, but it tries to put each task in the smallest worker node. Then, we can write the pseudo-code as presented in 2.2

```
for task in pendingTasks:
    for worker in sort(availableWorkers):
        if worker meets the requirements:
            place the task in the worker
            break
```

Listing 2.2: BFD pseudo-code

2.1.4 Spark

Spark [41] is an open-source framework for distributed data processing used to process big data. Although there are other frameworks for distributed data processing, like Hadoop [7] and Flink [6], Spark was chosen due to its popularity, since it allows a general-purpose programming language and has the ability to perform in-memory operations provided by the use of Resilient Distributed Datasets (RDD), which is one of its main features that will be explored later.

Resilient Distributed Datasets

A Resilient Distributed Datasets (RDD) [40] is a fault-tolerant distributed memory abstraction that allows to perform in-memory computations. It is implemented in Spark where we can apply multiple transformations to the data. RDD also allows users to reuse a RDD by persisting it in memory, then controlling the persistence, and to place partitions in a specific machine and then controlling partitioning, which allows placement optimizations. This way, it is possible to optimize operations like join by ensuring that the two datasets are hash-partitioned in the same way.

The main difference between RDDs and Distributed Shared Memory (DSM) systems is that RDDs can only be created using coarse-grained transformations, like map, filter and join, while DSM also allows fine-grained operations, like an update to a single value. Every time we apply a transformation to the RDD, a new RDD is created and the transformation is logged, which makes RDDs read-only collections.

Since we can only apply coarse-grained operations, RDDs are a good fit for parallel applications, since this type of application typically applies the same operations to multiple data items.

Although RDDs perform in memory, they can also perform operations on disk whenever the data is large enough not to fit in memory, which will provide similar performance to other data-parallel systems.

The fault tolerance is efficiently achieved by logging the transformations used to build the dataset instead of using the actual data. So, if a partition is lost, the RDD can recover that partition by applying the logged transformation to other RDDs.

Spark scheduler

Whenever the Spark scheduler has to run a job, it will have to create an execution plan. Spark will evaluate the RDDs lazily in order to find an efficient plan for user computation. To create the execution plan, Spark will look at the RDD dependencies and divide the job into multiple stages, each one containing multiple tasks depending on the number of partitions of the RDD. These tasks can be executed in parallel for each partition, and the output of a task will be the input of the next task in the same partition in the same stage. A stage can only start after all tasks from the previous stage have been completed, and whenever a stage ends, there will occur a shuffling operation, where all nodes send data to all other nodes. A representation of an execution plan can be seen in figure 4.

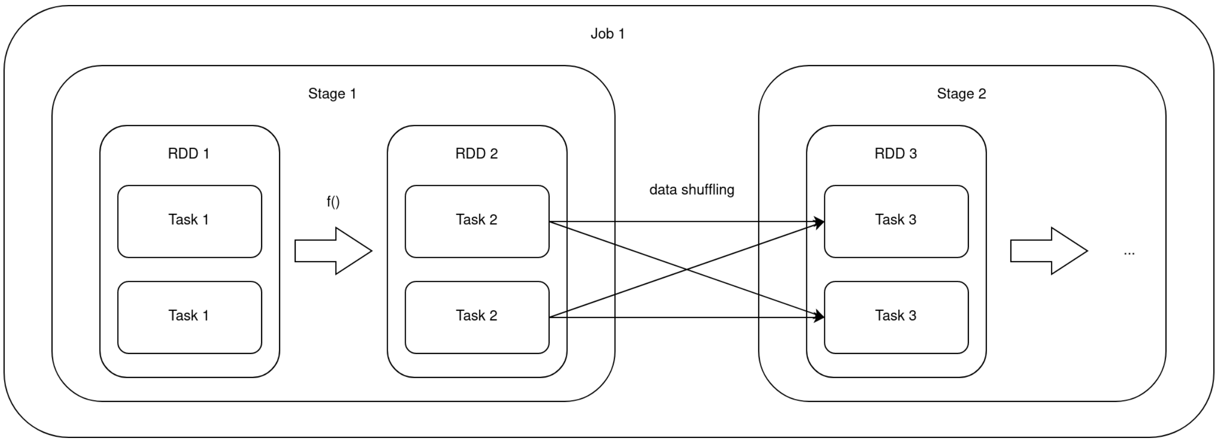


Figure 4: Spark execution plan example

This execution plan is represented in Spark as a Directed Acyclic Graph (DAG), where the vertices of the graph represent the RDDs and the edges represent the transformation to the RDDs. In detail, when a client submits a job, the driver will send the logical execution plan to the DAG scheduler. This logical execution plan is represented as a DAG composed of multiple transformations that will be applied to the data. Then, the DAG scheduler will divide the tasks into multiple stages considering their RDD dependencies. Finally, the DAG scheduler will send the resulting set of tasks to the task scheduler which will assign each task to an executor as previously mentioned.

When the application launches, Spark will launch multiple executors in various workers across the cluster to process the tasks. An executor is a process that runs in a worker node and is capable of executing a task. As previously mentioned, Spark has by default two different scheduling algorithms [2]:

FIFO and FAIR.

In the first one, the jobs will run in a first-in-first-out order and each job will try to use all available nodes. If a job doesn't require the entire cluster, the next jobs can run using the remaining nodes. Spark allows us to define the maximum number of nodes that each application may use.

For the second scheduling algorithm, Spark will assign jobs to the executors in a round-robin fashion, so that all jobs have an equal share of the cluster resources. This means that even if a long job is running in the cluster, a newly arrived short job will receive resources to run. The fair scheduler also allows us to group jobs into pools, where we can create multiple pools with different weights, in order to establish priorities between the jobs. This way, a job placed in a pool with a higher weight value will have a larger share of the resources.

2.2 Related work

2.2.1 Scheduling in heterogeneous environments

H. Li et al. [20] proposed EASAS which is able to reduce the energy consumption of Spark applications in heterogeneous environments under SLA restrictions to about 25-40%. This is achieved using a strategy table that stores historical data to evaluate the executors regarding their average energy consumption. There is a focus on prioritizing tasks with no historical data to detect unknown data in the strategy table. This algorithm greedily assigns tasks to the optimal executor and tries to put as many tasks as possible on it to reduce energy consumption.

The algorithm starts by iterating through the list of executors available and, for each one, will try to place as many tasks as possible, while the resources are sufficient, giving priority to tasks without information on the strategy table. During this process, the executors are sorted by their average energy consumption which was determined by evaluation criteria that uses the strategy table, and the tasks are organized in two sets, one for tasks that never run, which means that we have no information about them on the strategy table, and another one for tasks which we have information on the strategy table, sorted by their processing time in the executor we are analyzing.

W. Shi et al. [31] proposed TPCBFD and its energy-aware version EATPCBDF, which takes advantage of heterogeneous clusters by clustering the nodes and the tasks and assigning each task to the best fitting node, considering their historical data. EATPCBDF is capable of increasing the energy efficiency of Spark application by about 77% and at the same time increasing the SLA passing rate by about 14% in heterogeneous environments. Just like the previous algorithm, an energy model was proposed to measure energy consumption. In this work, they used HiBench as their testing benchmark and chose three workloads with different characteristics regarding CPU and IO usage, where they classified them as CPU-intensive, IO-intensive, or comprehensive.

TPCBFD works by first clustering the tasks based on their characteristics and also clustering the node performance. Here, they use k-means and a performance matrix that stores information for each host on multiple parameters about CPU and IO. Then, for every task, the executors are sorted by the task classification, and the task is placed on the first executor of the sorted list if it has enough resources. EATPCBDF works similarly to TPCBFD but when sorting the executors, it also takes into consideration the result of the effective energy consumption model they developed.

2.2.2 Machine learning and reinforcement learning in task scheduling

Recently, Mao et al. proposed Decima [22], a general framework that uses reinforcement learning and neural networks to learn complex scheduling algorithms without any human instructions, besides the specification of a high-level objective. This was possible by creating new reinforcement learning techniques capable of dealing with stochastic job arrivals. With this technique, it was possible to develop a model that learned how to reduce the average job completion time but can also learn to achieve other objectives by adjusting the reward during the training. This way, it is possible to use Decima to create a new scheduling algorithm just by specifying our goal.

Shi et al. [32] proposed a job scheduler for Spark in a hybrid cloud computing environment based on deep reinforcement learning. This algorithm aims to minimize the bounded slowdown and the total usage of the Spark cluster to increase performance and reduce costs. To do this, they designed a suitable reward function and used a simulated environment to achieve a reasonable scheduling policy. Their proposed DRL agent is capable of learning the characteristics of both the jobs and the environment to achieve the objective defined. Their experimental results show that this algorithm can improve the performance by 5.55% while reducing cluster usage by 13.9%. During their tests, they compared their algorithm against other scheduling algorithms besides the default from Spark: Best Fit Decreasing (BFD), Adaptive Executor Placement (AEP), and Greedy Iterative Optimization (GIO).

Berral et al. [5] proposed a scheduling algorithm that focuses on executing all the tasks in the minimum amount of nodes and turning off the idle nodes while also respecting the SLA restrictions. To achieve this, they trained a model that can predict whenever a reallocation of a job to another node will bring any energetic improvement. They used a simulated environment to evaluate their scheduler, which was able to reduce energy consumption while also respecting the SLA restrictions.

2.2.3 DVFS techniques in task scheduling

Other algorithms focus on DVFS techniques like FAESS-DVFS proposed by H. Li et al. [21] which reduce energy consumption by limiting the voltage and frequency of the CPU. In this work, the authors claim to achieve energy saving of up to 29.5% in Spark on YARN compared to the default algorithms, while also ensuring SLA restrictions. This algorithm is divided into two layers, the first is present in YARN, where a monitoring module is used to find the optimal CPU frequency that respects the SLA restrictions. The second layer is present in Spark, where the idle time of nodes is minimized by analyzing the DAG and assigning the heaviest tasks to nodes with higher computing performance, and by reducing idle times by decreasing the frequency of the nodes. This way, nodes that would finish their tasks before other nodes will have their CPU frequency increased, which results in a reduction of energy without increasing the job completion time.

In 2010, Wang et al. [37] proposed a Power Aware Task Clustering (PATC) algorithm that uses DVFS techniques. In this algorithm, the tasks are clustered by sorting the DAG edges by their communication costs and running them in the same computation node if the total power consumption does not increase due to the merging. They also proposed a technique to scale down the voltage of non-critical jobs to extend the execution time to its slack time, which corresponds to the time a job can be delayed without affecting the total execution time, and also scale down the voltage to the minimum whenever the node is executing a communication phase or is idle. According to their tests, this algorithm can save up to 39.7% of energy in a simulation.

Tang et al. proposed in 2014 a DVFS-enabled Energy-efficient Workflow Task Scheduling algorithm (DEWTS) [33]. This algorithm focuses on reducing schedule length and energy consumption as much as possible, and according to their tests, DEWTS can reduce energy consumption by up to 46.5%. This algorithm is divided into three phases. In the first phase, the algorithm uses the DAG to calculate the makespan and the deadline of all tasks. Then, it starts merging the processors in a way that the deadline is not violated. In this phase, the algorithm tries to place the tasks in N processors, and if the resulting makespan is lower than the deadline, it tries again with N-1 processors, until it finds the minimum number. Then, in the last phase, it will apply DVFS techniques to reduce the clock speed to extend the tasks and reduce the idle slots in a way that does not increase the total execution time.

2.3 Discussion

As we can see, most of the work about task placement is focused on reducing the energy consumption on heterogeneous clusters, so there is much work yet to be done around this type of scheduling policy. From these algorithms, we can see that to efficiently schedule tasks in this environment, we must take into consideration the various characteristics of both the jobs and the cluster so that we can provide the scheduler with more information and make better decisions in the scheduling algorithm. However, we should not overcomplicate the scheduler, since a more complex algorithm will also decrease the overall performance of the cluster. This way, we must analyze which characteristics will have the most impact on energy consumption and make the scheduler aware of those characteristics.

The use of machine learning models makes it possible to predict certain values, such as energy consumption, and determine the characteristics of the job, which can be effectively used on the scheduling policy to better achieve the goal. Without these models, we would need to resort to other methods in order to have this information, like running the jobs beforehand so that we can analyze and characterize them, making the use of these models a great help when deciding where to place each task.

The usage of reinforcement learning provides interesting results in a way that the scheduler can efficiently achieve the pretended results without any analysis of the job from the perspective of the programmer, making the development of a scheduler algorithm easier.

Algorithms that use DVFS techniques also perform well but require caution since we can end up slowing the execution in a way that increases energy consumption. This way, when using this technique, we need to find which tasks can effectively run slower without affecting the overall execution time. The usage of power capping shown in [11] also has excellent results since it provides a simple way of reducing energy consumption with minimal effects on the execution time.

This technique can also reduce idle times, where one task is finished earlier than others. By decreasing the voltage, we can extend the task duration and save energy without any effects on the total execution time.

Finally, task clustering can also reduce energy consumption by reducing communication costs. Since the tasks spend less time during communication by execution in the same worker, they can conclude the tasks faster. Yet again, we need to make sure not to overload the node in a way that the energy consumption increases instead of decreasing, which will require the scheduler to predict the amount of energy that its decision will produce.

Chapter 3

Analysis of Spark workloads

3.1 Experimental settings

In order to analyze the performance of the scheduling algorithms available in Spark, we decided to perform some initial tests where we analysed the behaviour of a few different workloads with different characteristics. We decided to use HiBench [15] as our benchmark, since it offers a large variety of workloads with options to customize the data size used in the benchmark. These workloads will be useful to analyze how the different properties of each job will be reflected in the algorithms and in the energy consumption. Another advantage of HiBench is that it can generate a report regarding CPU usage, memory usage, network usage, and other parameters that can be used to analyze the benchmark in the cluster. HiBench has also been a common tool for Spark workloads, as in [31].

Despite all the results that HiBench generates for us, it does not record any information regarding energy, which we will need to evaluate energy efficiency. So, to measure the energy usage in each workload, we will use Powerjoular [26] which is a tool capable of monitoring power consumption. Another option would be to use an energy model to approximate the energy consumption, however, we decided to use Powerjoular due to its better accuracy. This will allow us to make a realistic evaluation of the model compared to the use of an energy model that only gives us a theoretical value. However, it is worth noting that Powerjoular can only record the energy consumed by the CPU, so these results will not take into consideration any energy consumed by the network or any other piece of hardware.

This way, we needed to make some changes to HiBench to incorporate the values from Powerjoular. To do this, we created a script that initializes Powerjoular in every node of the cluster when the benchmark starts, and then we stop the measurements when the workload finishes. After that, we copy the results to the main node where we parse them and insert them into the report while it is created.

Regarding the workloads used, we decided to use PageRank, Sort, and TeraSort, since according to [31] they have different characteristics, where the first one is a CPU-intensive task, the second one is an

IO-intensive task, and the last one is a comprehensive task. With these three types of workloads, we will be able to test energy usage in various scenarios.

These tests were executed in a cluster composed of three machines equipped with an Intel(R) Core(TM) i5-10505 CPU @ 3.20GHz with 12 cores, 16GB of memory and a 256GB SSD. Also, for every test, we run the workload four times and record the average of each value in order to remove variance.

3.2 Comparison between scheduling algorithms

Firstly, we tested the differences between FIFO and FAIR with the workloads previously mentioned, and we recorded the total execution time and energy spent for every workload. We obtained the following results:

	Total energy (J)	Total time (s)
TeraSort	23203.44	242.853
Sort	10761.84	98.923
Pagerank	115165.8	971.254

Table 1: FIFO with 3 nodes

	Total energy (J)	Total time (s)
TeraSort	23921.3003	249.6092
Sort	11362.8562	101.9218
Pagerank	116098.4400	969.6776

Table 2: FAIR with 3 nodes

As we expected, the values for FIFO and FAIR are very close which confirms that the usage of these algorithms will have no impact on energy efficiency.

3.3 Analysis of each workload

Since HiBench gives us a detailed report about the utilization of every component during the execution of the workload of every worker, we decided to analyze in detail the characteristics of these workloads to find what are the differences between those tests that lead to different energy consumption. Since the results of FIFO and FAIR were very similar we will only consider the tests made using the default FIFO scheduler.

After concluding each test, we calculated the average CPU and memory usage, as well as the average number of IO operations. These results are presented in the table:

	Average CPU utilization (%)	Average memory utilization (%)	Average number of IO operations
TeraSort	34.2636	49.5956	4424.6894
Sort	38.0322	39.9165	8783.1263
Pagerank	48.3792	59.7733	210.0785

Table 3: Resources usage

According to the results shown in table 3, we can see that TeraSort has the lowest average CPU usage and high average memory usage, so it makes sense to classify it as a CPU-intensive task, whereas Pagerank has the highest average CPU usage and also the highest memory usage.

Looking at the table 1, we can calculate the average power that each workload requires.

	Average Power (w)
TeraSort	95.5452
Sort	108.79
Pagerank	118.5743

Table 4: Average power using FIFO

Comparing the tables 4 and 3, we can see that the higher the CPU usage, the higher the average power required. This makes sense because Powerjoular will only register the power required for the CPU and the GPU, which means that the power spent by other components is not measured. Since these workloads do not use GPU, these results only represent the CPU consumption.

The results also show that the CPU usage will be the lowest when the network usage is the highest, as we can see in the firgures 5 and 6, which show a heatmap for CPU usage and network, respectively. This leads to the conclusion that when an IO operation occurs, the CPU will remain idle. So, if the data is not well distributed across the hosts, it will greatly increase the execution time. Once again, during IO operations, the energy consumption will be minimal due to the low CPU usage.

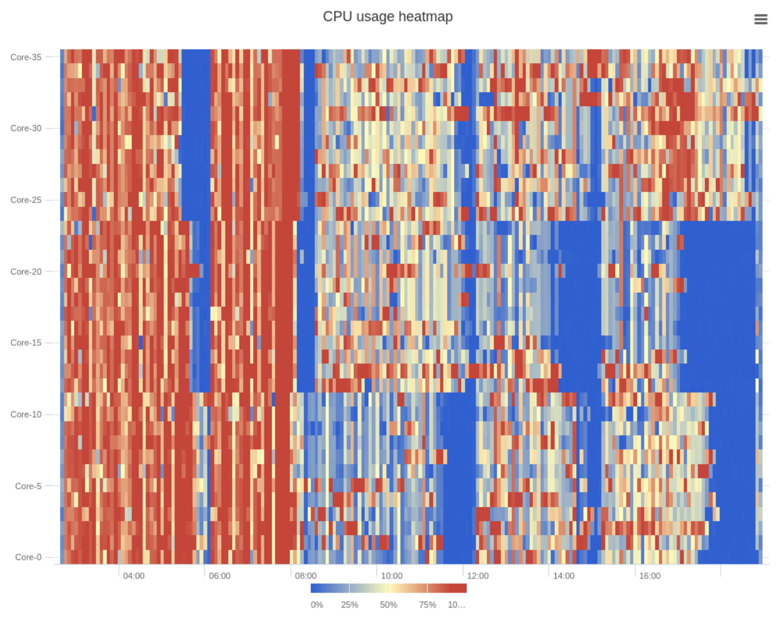


Figure 5: CPU usage heatmap of Pagerank

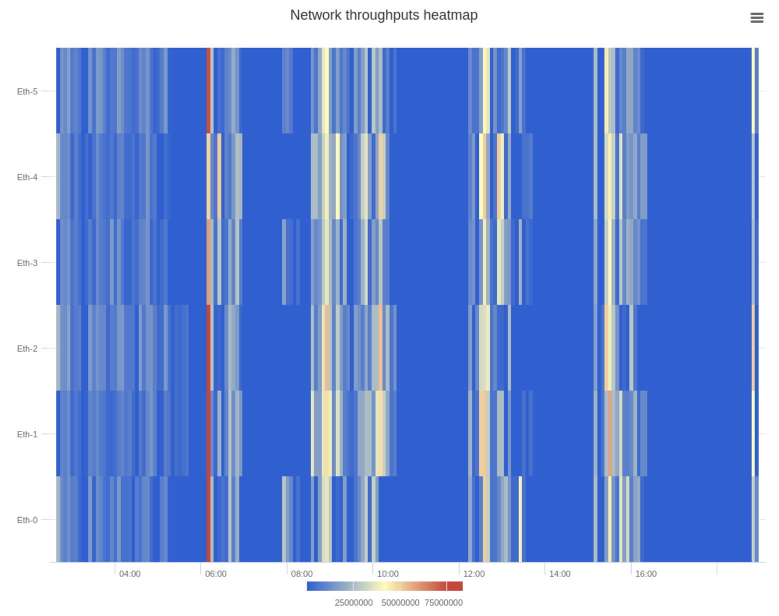


Figure 6: Network usage heatmap of Pagerank

If we compare the network usage in figure 6 with the DAG logical plane presented in figure 7, we can see that these high network values correspond to shuffle operation, where each cluster node sends data to the other cluster nodes.

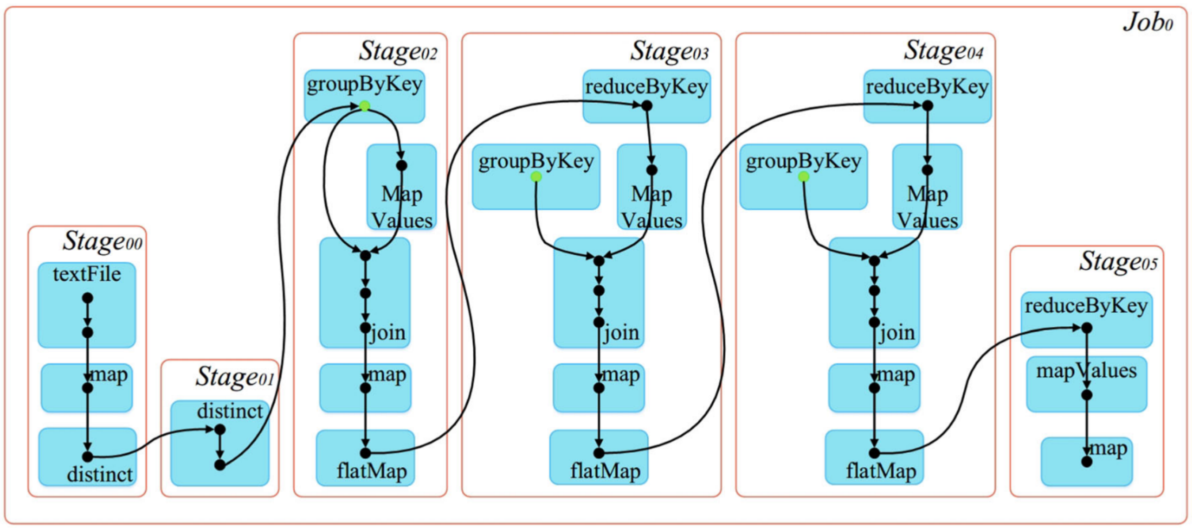


Figure 7: DAG logical plan of Pagerank [20]

Finally, when we represent the power consumption in a heatmap, as presented in figure 8, we can see that the energy usage is minimal when the CPU usage is low and the network usage is high, which corresponds to the blue areas. In addition to that, we can also see that the power consumption is at its maximum when the CPU usage is also at maximum since the red zones overlap, which is to be expected since the tool used to measure the power consumption only records the power consumed by the processor.

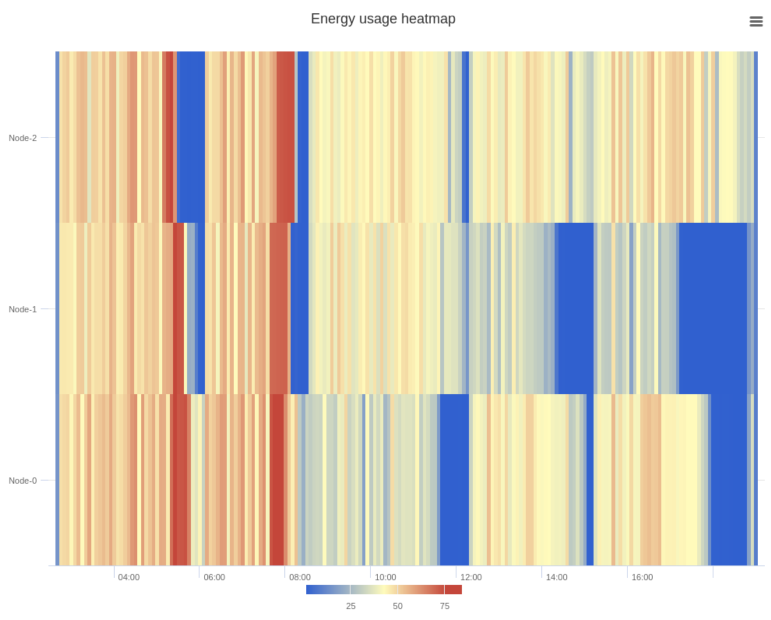


Figure 8: Energy consumption heatmap of Pagerank

3.4 Testing the number of workers

Finally, after confirming that the choice of the default algorithms would not affect our results, we tested if the reduction in the number of workers for each job would impact the results. With this test, we want to confirm if the correct choice of the number of resources benefits energy consumption.

	Total energy	Total time	Average Power
TeraSort	21720,4067	288,7796	75.2145
Sort	10612,5063	121,2202	87.5473
Pagerank	118219,5552	1300,877	90.8768

Table 5: FIFO with 2 nodes

We can see the results of using only two workers in table 5. We can see that the reduction in the number of workers in the cluster reduced energy consumption for all workloads compared to the results obtained in table 4. So, we can conclude that the choice of the number of workers will impact energy consumption. In this case, the execution time increased, but the power consumption was much lower, which means that if this new execution time was lower than a possible deadline, it would be preferred over the previous execution with three workers.

3.5 Energy consumption analysis

After confirming that the usage of the different scheduling algorithms implemented in Spark produces similar results regarding energy consumption, we decided to test how changing the number of cores of each executor and the frequency of the processor affects the energy consumption. We expect energy consumption to increase exponentially with the increase in the frequency of the processors, and the execution time to decrease exponentially with the increase of the frequency, as we mentioned in section 2.1.2.

The impacts on energy consumption by changing the number of cores is hard to predict. In theory, the execution time decreases with the increase in the number of cores, and consequently, the energy consumption should decrease. However, the use of more cores increases the load of the CPU so it will require more power. Furthermore, the increase in the number of cores might even decrease the CPU due to memory restrictions, where the CPU might not be able to receive more data.

We also want to test how these factors impact the energy consumption on the various workloads. A workload that puts a heavy load on the CPU might be affected differently by the number of cores than a workload that heavily relies on IO.

This way, we re-run every workload in a cluster composed of a single node to analyse the impact of each configuration. We recorded the total energy consumption using Powerjoular and the execution time of each workload and calculated the average of four executions to determine the value. We are assuming homogeneous cores, so we will not consider efficiency cores and performance cores as we see in modern processors.

Regarding the values used for each parameter in the analysis, we can use up to 12 cores and the frequency ranges from 0.8 GHz to 4.6 GHz for this CPU model. So we tested every number of cores (from 1 to 12), and five different frequencies: 0.8 GHz, 1.2 GHz, 2.3 GHz, 3.4 GHz, and 4.6 GHz. The profile used for each workload was: 'large' for Pagerank and Terasort, and 'huge' for Sort.

- Why we made these tests
- What workloads will be used
- Explain why we are analyzing the variation in the energy consumption caused by the CPU frequency and number of cores
- What values we will measure
- The configuration of each test
- A description of the machines

3.6 Energy consumption results

Starting with Pagerank, we can see in figure 9 that the energy consumption of the cluster for this workload increases exponentially with the increase of the frequency for all number of cores, as expected. He can also see that for higher frequencies, a higher number of cores will consume more energy,

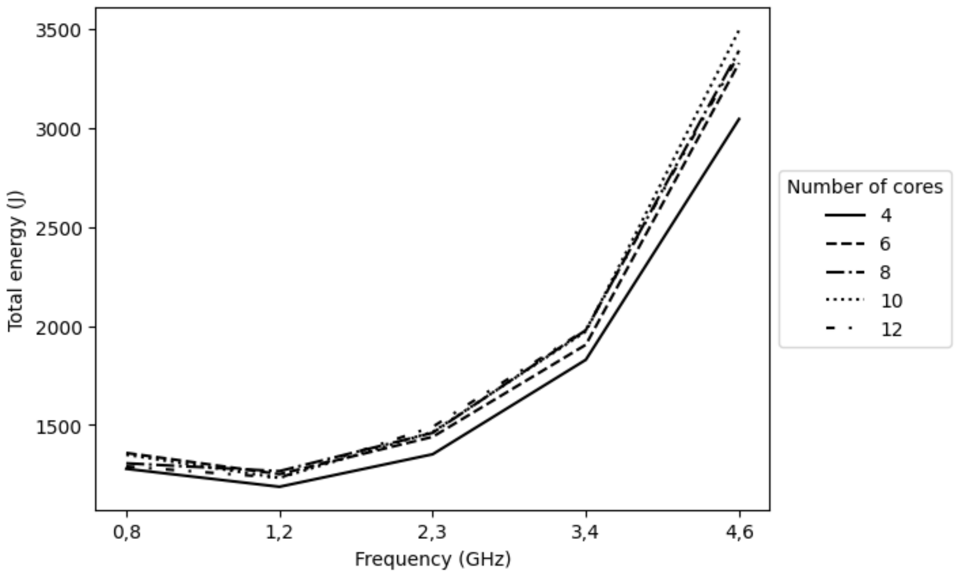


Figure 9: Total energy of Pagerank

In figure 10, we can see the results for the execution time for Pagerank. Once again, as expected, we can confirm that the execution time of the workload decreases logarithmically with the increase of the frequency. Also, a higher number of cores will consume more energy.

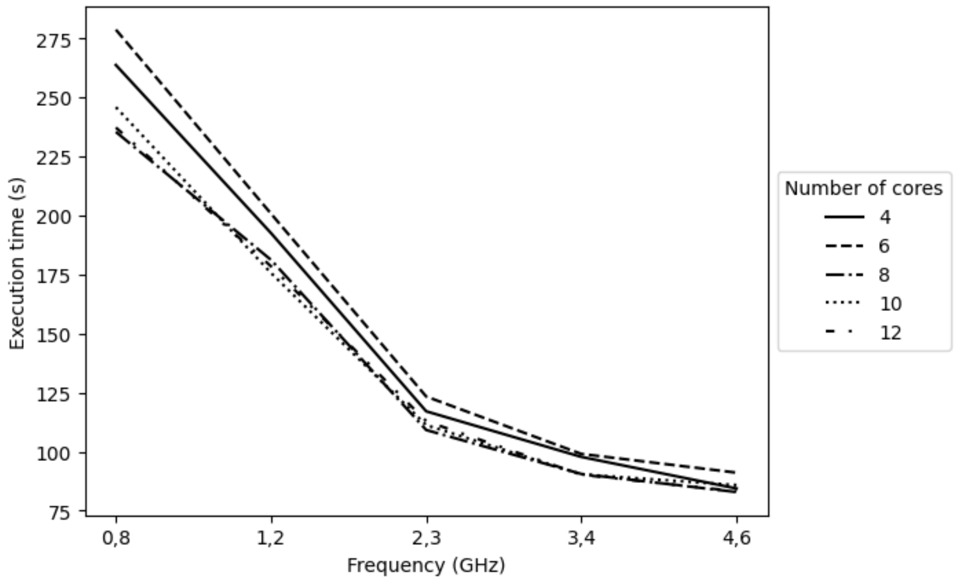


Figure 10: Execution time of Pagerank

The results presented in figure 11 show how the number of cores affects the energy consumption for various frequencies. These results show that in this workload, for lower frequencies energy consumption decreases until four cores and maintains after that. For higher frequencies (3.4 GHz and 4.6 GHz), the results show that the energy consumption increases with the number of cores.

One interesting result is that for every frequency, using two cores will always be better than using a single core, and for lower frequencies (0.8 GHz and 1.2 GHz) using a single core will be the worst option. When we increase the number of cores in an executor, we are also putting more load into the processor, which makes it require more energy to operate. When we go from one core to two cores, the decrease in the execution time must be so high that it contradicts the tendency to increase power consumption.

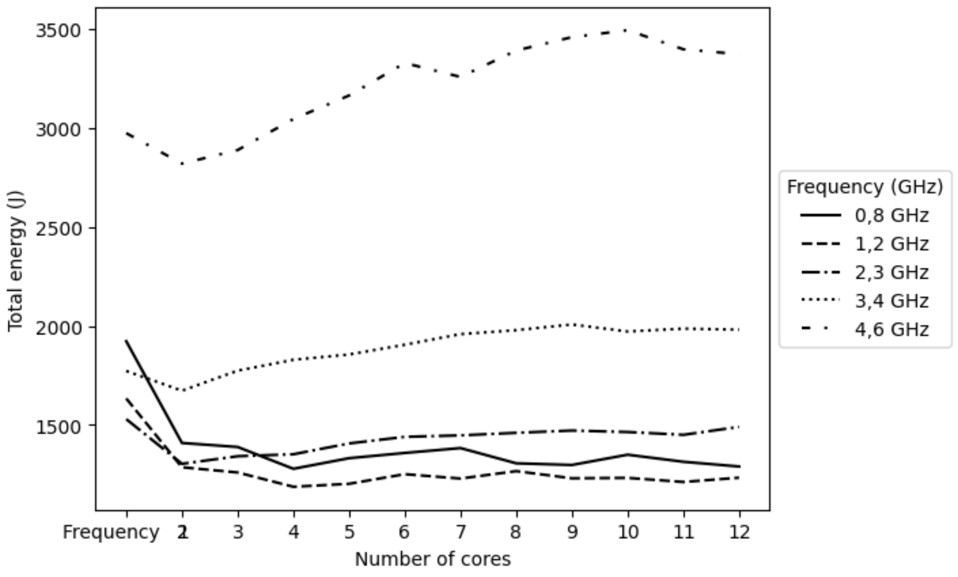


Figure 11: Total energy of Pagerank per number of cores

Finally, figure 12 shows how the execution time of the workload Pagerank varies according to the number of cores for the frequencies tested. As we can see, the execution time decreases with the number of cores used for every frequency. Although, after four cores the decrease is practically unnoticeable. A possible explanation is that when using more cores, the improvement in the execution time of a workload will be limited by other factors such as the bandwidth until it eventually reaches its limit. As we predicted previously, the decrease in the execution time from one core to two cores is the highest, and that is why the total energy always decreases from one to two cores. In lower frequencies (0.8 and 1.2 GHz), the decrease in the execution time from two to four cores is still noticeable and can explain why in the previous test 11 we can observe a decrease in the energy consumption in this configuration.

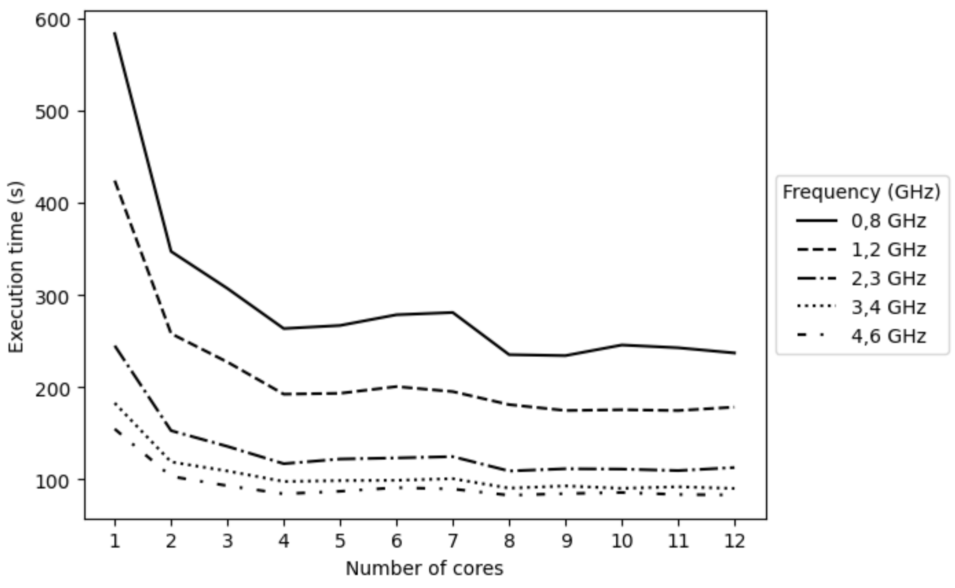


Figure 12: Total energy of Pagerank per number of cores

The results for the total energy consumption on Terasort 13 show the same behaviour as the ones for Pagerank, where the energy increases exponentially when the frequency increases.

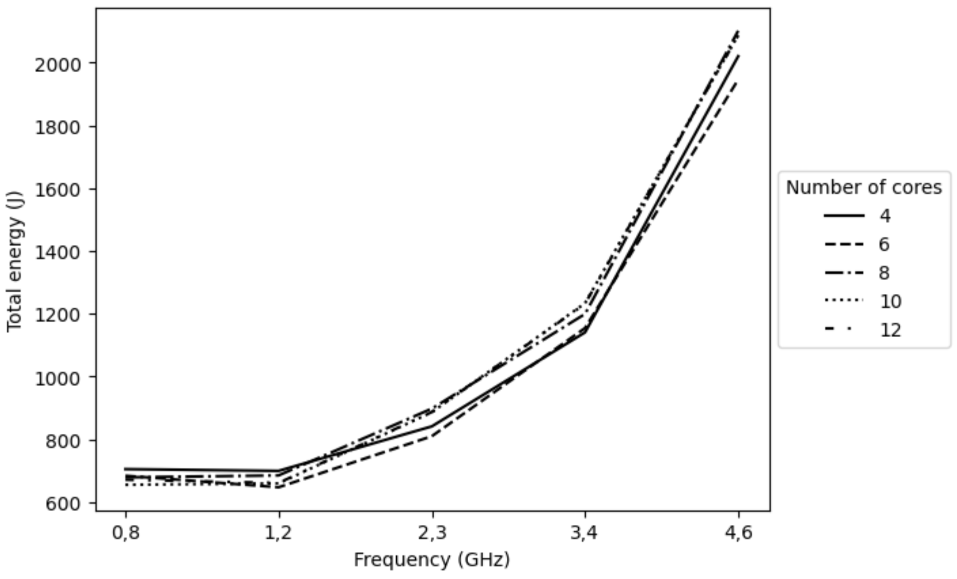


Figure 13: Execution time of Terasort

For the total energy results presented in 14, once again we observe the same results as the ones on Pagerank, where the execution time decreases logarithmically when the frequency increases.

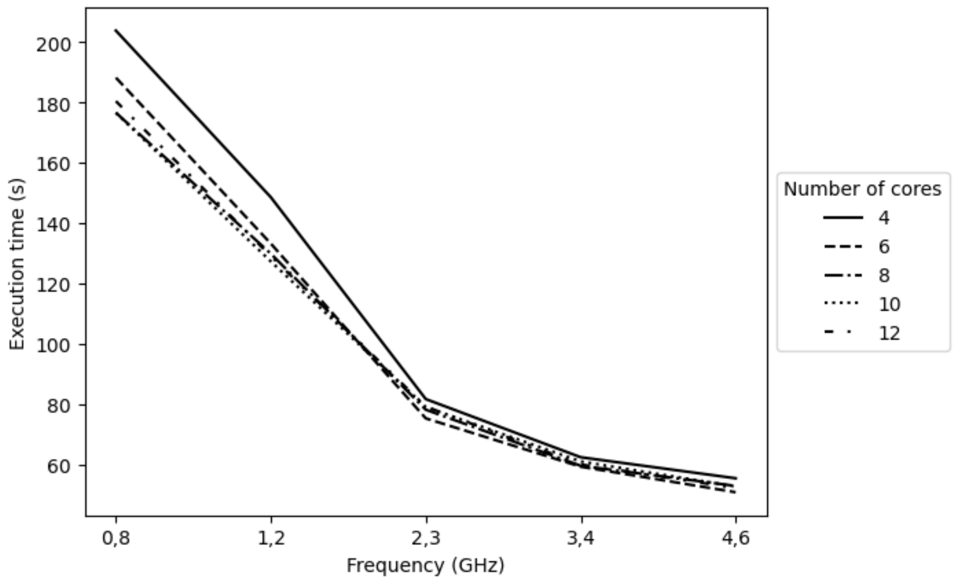


Figure 14: Execution time of Terasort

Regarding the effects of the number of cores on the total energy consumption 15, we can see that for this workload the energy consumption increases slightly with the increase in the number of cores for frequencies between 2.3 GHz and 4.6 GHz, with the exception of lower frequencies (0.8 GHz and 1.2 GHz), where the energy consumption decreases slightly with the number of cores. These results are justified due to this workload being classified as a comprehensive workload, requiring less CPU than the previous one.

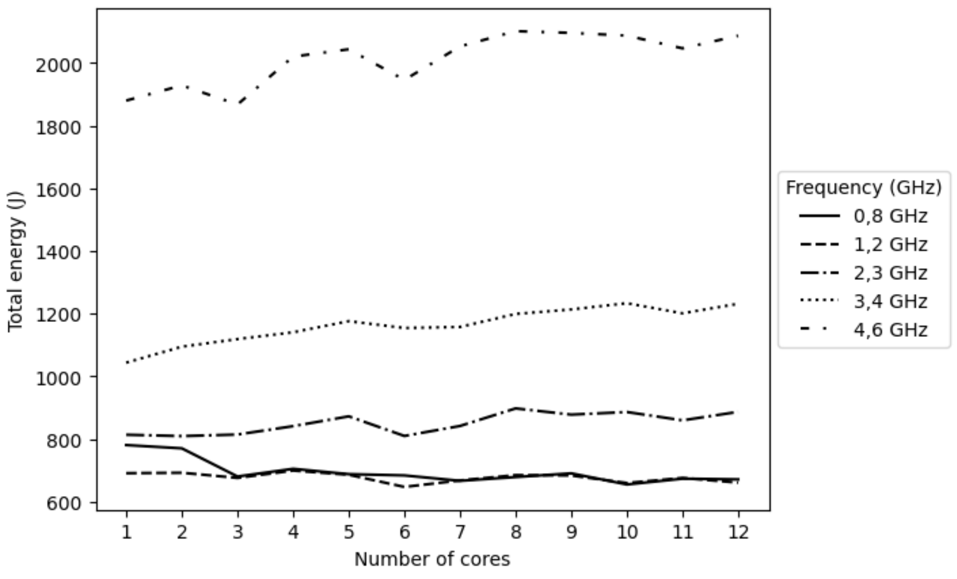


Figure 15: Total energy of Terasort per number of cores

In figure 16 we can see the relation between the execution time and the number of cores for every tested frequency. Like the results shown for the Pagerank workload 12, the execution time decreases logarithmically as the number of cores increases, for all frequencies.

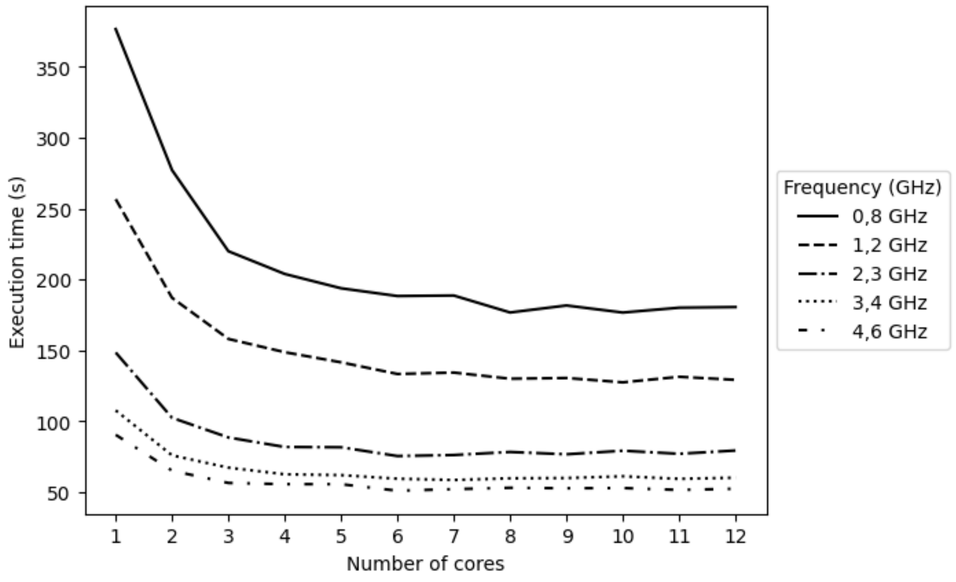


Figure 16: Execution time of Terasort per number of cores

Moving on to the Sort workload, figure 17 shows the result for the total energy for every tested frequency on various cores. Like the previous workloads, the total energy increased exponentially with the frequency increase on every number of cores tested.

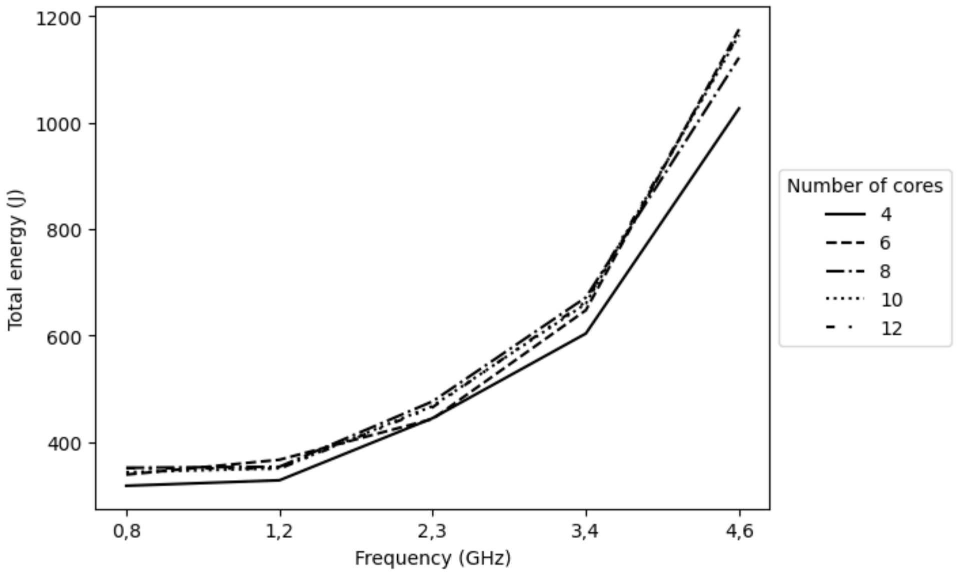


Figure 17: Total energy of Sort

For the execution time 18, we can see once again the same behaviour as the same test on the previous workloads, where the execution time decreases logarithmically as the frequency increases, for every tested number of cores.

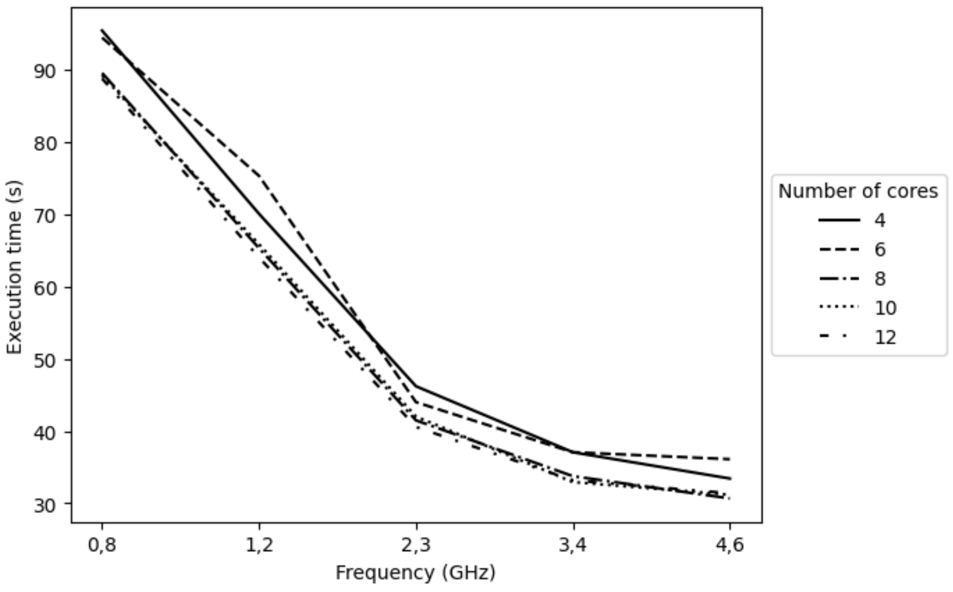


Figure 18: Execution time of Sort

In figure 19 we can see the relation between the total energy and the number of cores for every frequency tested. For the lower frequencies (0.8 GHz and 1.2 GHz), the total energy remains roughly the same after two cores. For higher frequencies (3.4 GHz and 4.6 GHz), it increases from two cores to six cores and maintains after that. In addition to that, in every frequency tested, using a single core will require more energy than using two cores.

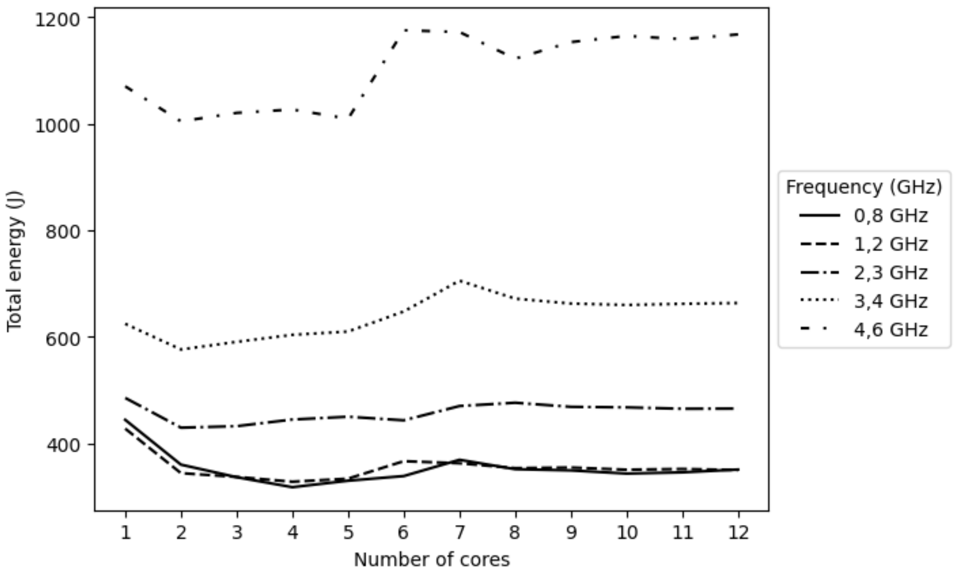


Figure 19: Total energy of Sort per number of cores

For the last test 20, we observe the same behaviour as the last two workloads, where the execution time decreases logarithmically as the number of cores increases.

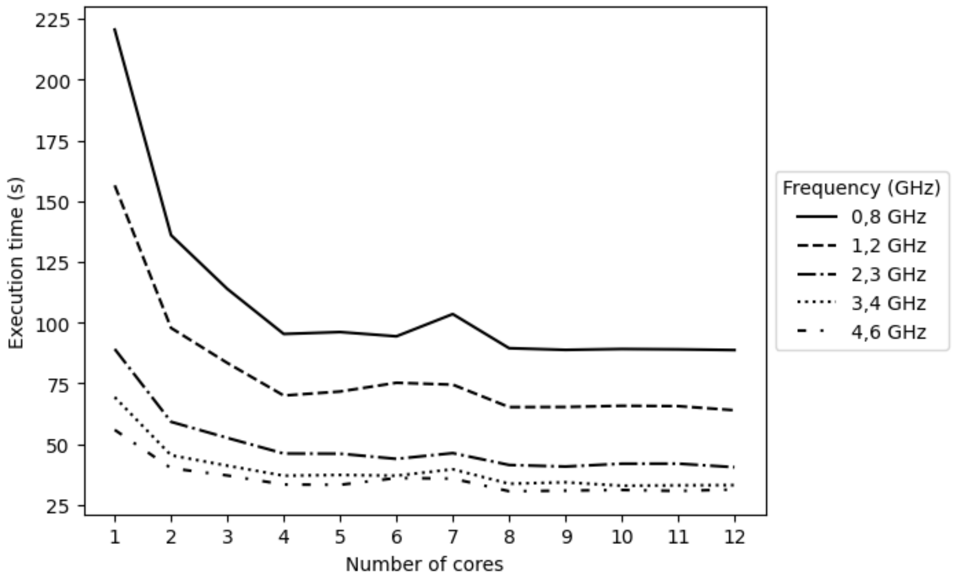


Figure 20: Execution time of Sort per number of cores

3.7 Discussion

After analysing these results we can take many conclusions. The first one is that as expected, the energy consumption increases exponentially with the increase of the CPU frequency and the time decreases logarithmically with the CPU frequency in all tests.

In general, the total energy consumption decreases with the number of cores for lower frequencies, while for higher frequencies it tends to increase.

The execution time decreases in all workloads with the number of cores until four cores and stabilizes after that.

Although the workloads usually end early with higher frequencies and a high number of cores, this is very inefficient when looking at the energy consumption results, since it is possible to achieve a similar execution time with a lower frequency and a higher number of cores with much lower energy consumption.

Finally, in some cases the total energy consumption increases with the number of cores, however, in both Pagerank and Sort workloads it is much more efficient to use two cores than to use one single core in both total energy consumption and total execution time for every frequency tested. On Terasort workload, there is only a slight increase. This indicates that it is possible to decrease the total energy consumption if we decrease the number of hosts that are only processing a single task and merge that task with another host that is not fully utilized.

This also shows us that using fewer cores will not always be better, just as decreasing the frequency will not always be the best option since it increases the execution time.

Chapter 4

EASAHUM

4.1 General idea

EASAHUM focuses on reducing energy consumption resorting to two main strategies. Firstly, it will reduce the number of hosts that are unnecessary for the execution of the workload without affecting the execution time, by turning off hosts that are not running any tasks. This means that when a host is not executing tasks, it will be suspended or shut down to save energy. The second main strategy consists of reducing the amount of time when the host is underutilized by rearranging the execution order of the tasks. An underutilized host is a host that is not being used most efficiently, in this case regarding energy consumption. As we saw previously in section 3.6, running a small number of tasks in a host is very inefficient in terms of energy consumption, especially when comparing running a single task to two tasks in parallel, so EASAHUM will focus on adjusting the order of execution and where each task runs to reduce the situations where we have hosts running very few unique tasks. By reducing the amount of underutilized hosts, EASAHUM will also potentially reduce the up-time of some hosts which also helps reduce energy consumption.

As in other list scheduling algorithms, this one will be divided into two parts [34]. In the first part, tasks that are ready to run are sorted according to specific criteria, whereas in the second part, EASAHUM will use the list of tasks sorted previously and decide for each task which host will run it, using the order in which the tasks were sorted. During this second part, it is not mandatory to schedule every task, since there might not be enough resources for every task, or the scheduler might even decide not to schedule a specific task at that moment.

In the first part, where the tasks are sorted, EASAHUM will prioritize the execution of tasks that will free new tasks, so that the cluster always has tasks ready to execute, which allows EASAHUM to have a larger number of options. Moreover, EASAHUM will also prioritize longer tasks, but with less priority than the previous criteria, to keep shorter tasks to the end of the job execution. This will allow the scheduler to use shorter tasks to fill empty slots caused by larger tasks and also increase the machine utilization in the later phases of execution since there will be less variance between the finishing times of those tasks.

To determine if the execution of a task t will free new tasks, EASAHUM will calculate a score based on the number of tasks that t frees (i.e. the number of childs of t) and the number of total dependencies that each child has. Ideally, the algorithm would consider the exact number of unique child dependencies, however, since this is a costly operation, the algorithm will not be able to count unique child dependencies. Instead, it will use the average of the child dependencies which is a simple calculation and offers excellent results, especially on workloads where most childs have the same dependencies, which happens in our study case. This way, EASAHUM will use the equation 4.1 to calculate the score of each task.

$$Score(t) = \text{num\_childs}(t) - \frac{\sum_{\text{child} \in \text{childs}(t)} \text{num\_dependencies(child)} - 1}{\text{num\_childs}(t)} \tag{4.1}$$

This way, the result of equation 4.1 will be positive when the number of new tasks freed after the conclusion of task t is higher than the number of tasks required to free those tasks, and the result will be negative otherwise.

When choosing the host where each task will run, EASAHUM will make different decisions depending on whether there are enough tasks to keep the cluster occupied or not, since it is possible to save the most amount of energy at the end of a stage, when there are idle resources. This algorithm will consider that having enough tasks to maintain the cluster occupied is the same as having more tasks than resources, where a resource is the core of a host that can run a single task. Although this is a very simple condition, it offers good results in most cases. This condition can be improved since there will be cases where there are fewer tasks than resources, but there are still enough to keep the resources occupied, as we can see in figure 21.

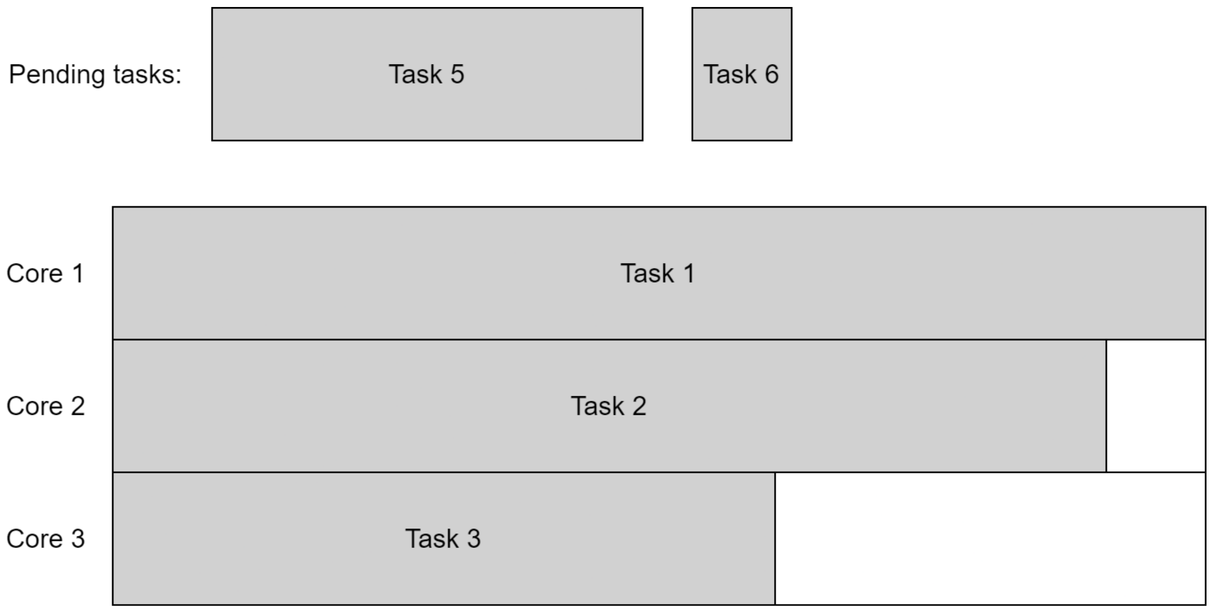


Figure 21: Enough tasks to keep the resources occupied

Figure 22 shows another possible scenario, where there are more tasks than resources but they are still not able to keep the machine fully occupied.

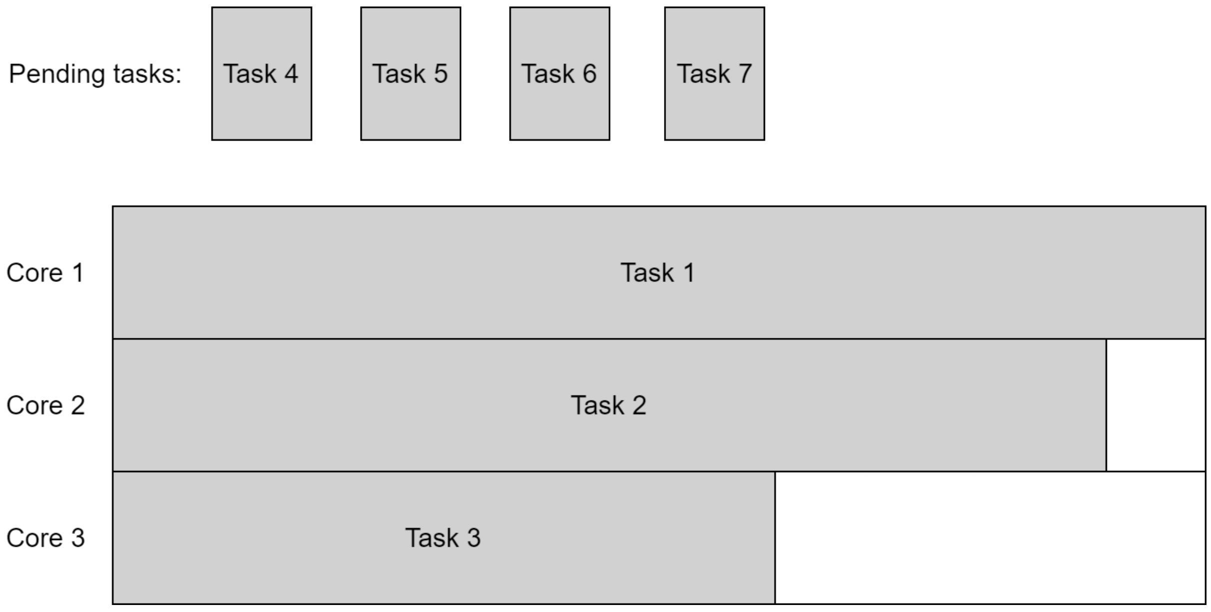


Figure 22: Not enough tasks to keep the resources occupied

As we can see in the previous examples, the condition will fail to find exactly if the tasks that are yet to be executed can keep the resources occupied in most cases. However, changing EASAHUM to consider those cases would have a big impact on the time complexity of the algorithm since this operation is made for every try to schedule a task. So, we will keep that condition since its objective is to set a limit on when

EASAHUM should start performing more computationally heavy operations to save energy, and not to find exactly if we can have a perfect scheduling.

When the amount of tasks ready to schedule is not enough to keep the resources occupied, according to the condition previously mentioned, EASAHUM will find among the available hosts the one whose time where the host is underutilized is closer to the expected runtime of the task, and this one will be called the best fit. The time when the host is underutilized will be called idle time. The objective of finding the best fit is to reduce host underutilization as much as possible, which happens when there are not enough tasks to maintain all resources occupied. This way, EASAHUM will only calculate the best host when there are few tasks ready to execute in the cluster, which will greatly decrease the computation time of the algorithm.

When using this strategy, there may be some problems. If EASAHUM is allowed always to use the best fit, then it may use a host that could be turned off to save energy and schedule that task in a host with similar idle time. An example of this problem is shown in figure 23. In this case, the host "worker1" could be turned off before 200 seconds and place the remaining tasks on host worker2 at around 300 seconds.

So, to fix this problem, we can limit EASAHUM to only use the best fit if its placement does not increase the underutilization of that machine. However, if at any point the new tasks are big enough to increase underutilization, the algorithm would delay the scheduling of those tasks until all other tasks are finished, which would increase the energy consumption. An example of this problem is shown in figure 24, where around 300 and 400 seconds, those three tasks were only executed after the other tasks finished, which should not happen since those are not dependencies for the three tasks.

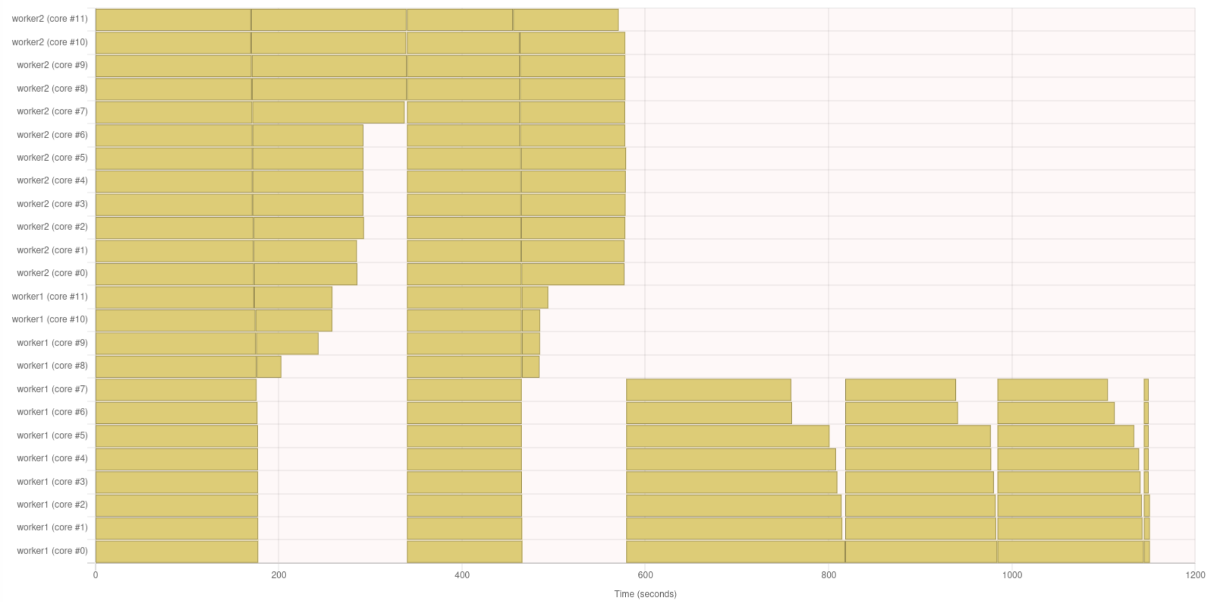


Figure 23: Example of problems when always using best fit

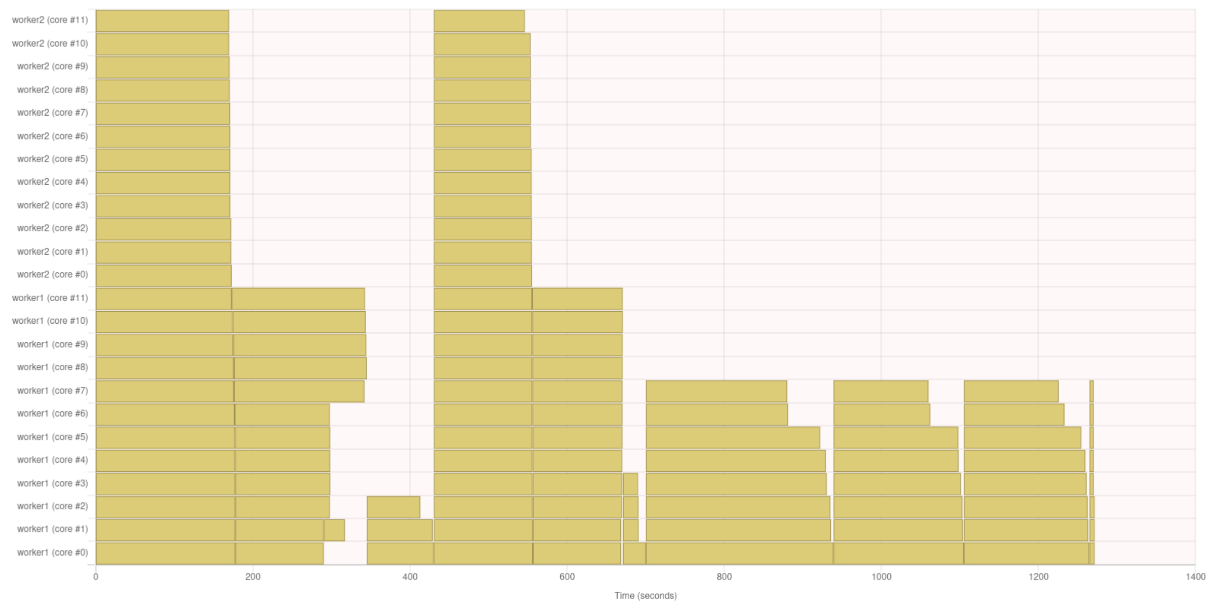


Figure 24: Example of problems when tasks are too long

So, the solution to this is to mark machines that have a low underutilization whenever there are more available tasks than unmarked cores, where if a host is marked, then EASAHUM can only schedule tasks on it if that machine already has a task that is expected to finish later. The objective is that while the amount of tasks ready to schedule decreases, the number of resources turned on also decreases.

The process of marking a host will happen before choosing the host to schedule the task and happens while there are more tasks than resources from unmarked hosts, but with the condition that there must be at least one unmarked host as long as there are hosts turned on so that it is possible to schedule tasks.

In case the best host does not meet the requirements, the scheduler will try to place the task in a host whose underutilization does not increase due to the placement of the task. Finally, EASAHUM can also choose not to schedule the task if there is not any host that follows the previous condition, or if there is not any host available. In this case, the task will stay in the list of tasks that are ready to be executed and the scheduler will try to schedule them again when a new resource is available i.e. when a task ends.

In short, when there are few tasks on the cluster the algorithm will make the following decisions after calculating the best host among the available hosts at the moment:

1. If the best host is not marked, EASAHUM will use the best host.

2. If the best host is marked but the underutilization does not increase when the task is placed there, EASAHUM will schedule the task in the replacement host.

3. If there is a host whose underutilization does not increase when the task is placed there, EASAHUM will schedule the task in that host.

4. If there are no hosts to replace the best host, EASAHUM will not schedule the task.

When there are many tasks in the cluster, we will use the first host that is free since it is a simple operation. Moreover, in the case where there still are a lot of tasks to schedule and we have hosts turned off, the scheduler can turn on a host and use it if needed. So, when there are many tasks in the cluster, the algorithm makes the following decisions:

1. If there is an available host, EASAHUM will schedule the task in that host.
2. If there is no available host but there are hosts turned off, EASAHUM will turn on one host and schedule the task in that host.
3. If there is no available host and all hosts are turned on, EASAHUM will not schedule that task.

4.2 EASAHUM

In this section, we are going to explain EASAHUM in detail. As we explained previously, the algorithm is divided into two parts. The first part, where the tasks are sorted, is presented in 1 while the second part, where the machine for each task is chosen, is presented in 4 and 6.

As we can see in the algorithm 1, the scheduler will first prioritize tasks that have a positive score over tasks with a negative score. This score is calculated according to equation 4.1 which was already explained. When both tasks have a positive score or any of the tasks have a negative score, EASAHUM will prioritize the tasks based on the predicted execution time. Finally, if two tasks have similar execution times in this case, the scheduler will use the task name to set an order. So, the order of tasks will be defined by the following criteria of prioritization:

1. Long tasks with a positive score.
2. Short tasks with a positive score.
3. Long tasks with a negative score.
4. Short tasks with a negative score.

Algorithm 1 Condition to sort two tasks

Input: t_1 and t_2 tasks

Output: True if t_1 should be executed before t_2, and False otherwise

```
1: if $getTaskScore(t_1) > 0$ and $getTaskScore(t_2) \leq 0$ then
2:     return $true$
3: else
4:     if $getTaskScore(t_2) > 0$ and $getTaskScore(t_1) \leq 0$ then
5:         return $false$
6:     else
7:         if $predictTime(t_1) == predictedTime(t_2)$ then
8:             return $t_1 < t_2$
9:         else
10:            return $predictedTime(t_1) > predictedTime(t_2)$
11:        end if
12:    end if
13: end if
```

The prediction of the execution time of a task will not be explored in this dissertation. However, a way of predicting the execution time of a task is using machine learning techniques like in [27], where they were able to achieve low error percentages for the majority of the tasks. Later we will evaluate how the degree of uncertainty affects the performance of the algorithm.

The algorithm for finding the best fit for a task is presented in 2, which goes through the list of hosts (lines 3-11) and calculates the idle time of each host (line 5) using the algorithm 3, in order to find the host with the closest idle time to the task expected runtime.

Algorithm 2 Find best fit

Input: t task and $hosts$ list of Hosts

Output: Host from $hosts$ with closest idle time to t expected runtime

```
1:  $best\_time \leftarrow +\infty$
2:  $best\_host \leftarrow$ None
3:  while $hosts$ is not empty do
4:      $host \leftarrow$ next Host on the list of Hosts $hosts$
5:      $idle\_time \leftarrow$ expected idle time for Host $host$
6:      $diff \leftarrow |predicted\_time(t) - idle\_time|$
7:      if $diff < best\_time$ then
8:          $best\_time \leftarrow diff$
9:          $best\_host \leftarrow host$
10:     end if
11: end while
12: return $best\_host$
```

The algorithm 3 calculates the idle time of a host. When doing this operation a Host h three scenarios can happen:

1. All resources of the Host h are used at the start of the algorithm.
2. Host h has at least one resource occupied, but it is not fully utilized.
3. Host h is not running any task.

In this calculation, the time where a Host h is a free resource to execute a task will be referred to as $lowest$ and the expected time where the number of resources of the host increases will be referred to as $highest$. Also, the idle time will be calculated as the difference between $highest$ and $lowest$, as shown in the equation 4.2.

$$Idle\ time = |highest - lowest| \tag{4.2}$$

In the first scenario, $lowest$ will be the time when the first task running on h ends, which will be the first task to finish (lines 7-13), and $highest$ will be the time when the last task running on h ends, which will be the last task to finish (lines 18-24). In figure 25 we can observe an example of this scenario. In this example, the value of $lowest$ will be 260, since it is the time when the first task is expected to finish, and the value of $highest$ will be 430 since it is the time when the last task is expected to finish. This way, the idle time for the host in this example will be 430 - 260 = 170.

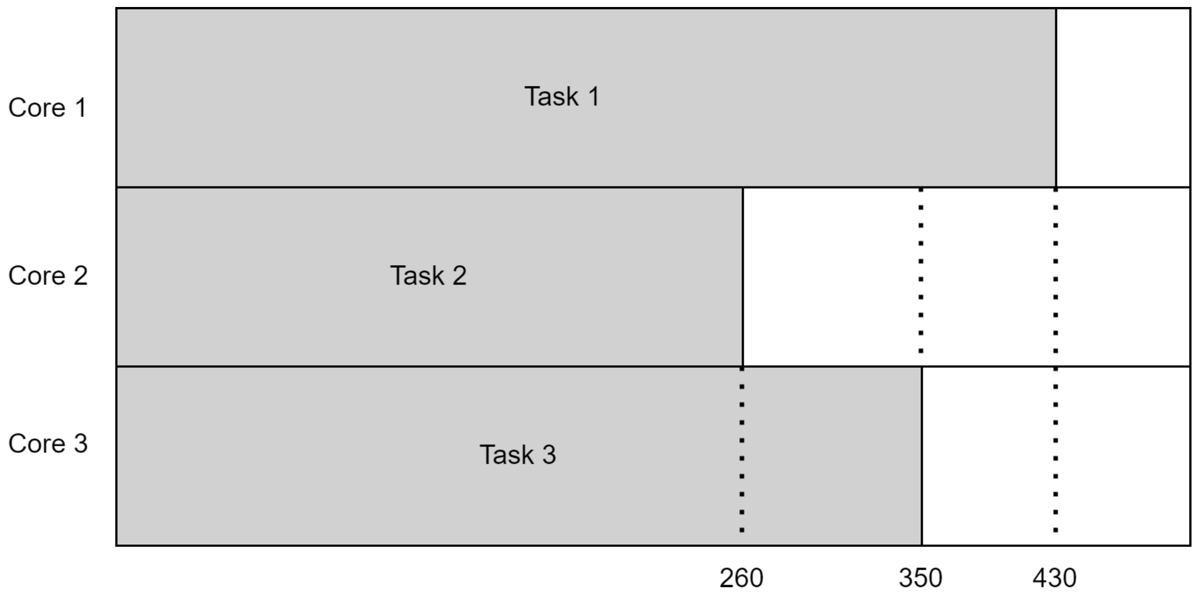


Figure 25: Example for scenario 1

In the second scenario, h has free resources, so EASAHUM will consider $lowest$ as the current time (lines 4-5), and $highest$ will be calculated in the same way as the previous scenario. Figure 26 shows an example of this scenario, where one of the cores is available. This way, the value of $lowest$ will be 0, considering we are at the start of the execution, and the value of $highest$ will be 430 since it is the time when the last task is expected to finish. So, the value of the idle time for this example will be 430 - 0 = 430.

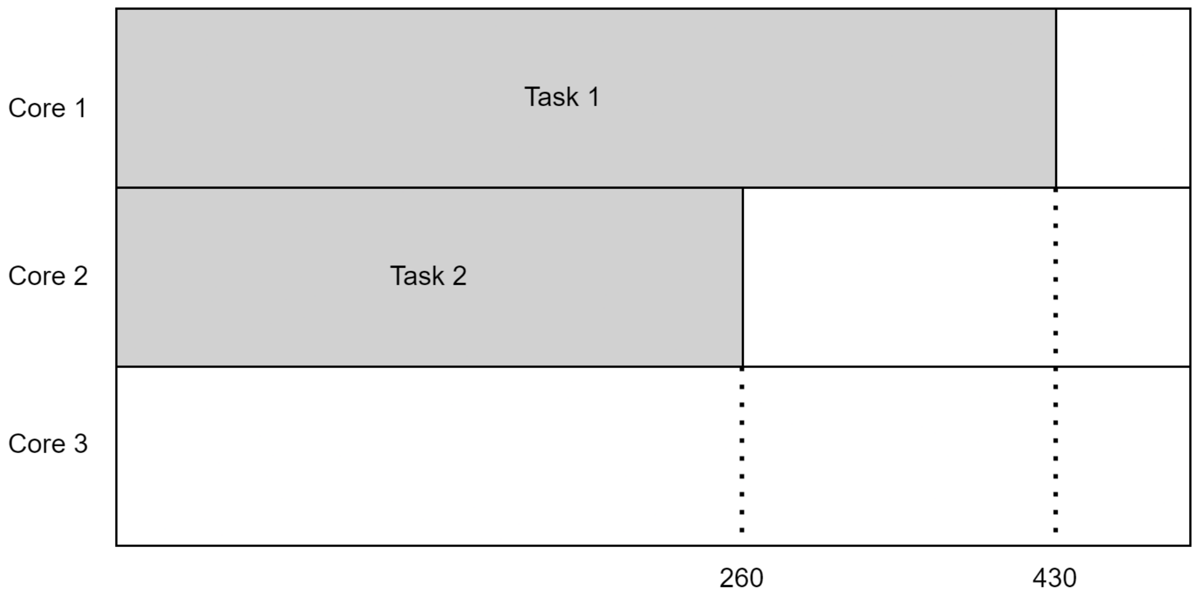


Figure 26: Example for scenario 2

Finally, in the third scenario, h is not running any task, so we will use the current time as the value to $highest$ instead of infinite (lines 15-16), and the $lowest$ will be the same as in the second scenario. This way, the idle time for this machine will be zero which allows EASAHUM to prioritize other hosts that are already in use and also allows EASAHUM to shut down the current h in the case it is not used. It is worth reminding that in this case, the host can still be used if there are no other options.

Algorithm 3 Calculate idle time

Input: h Host

Output: Approximated idle time for Host h

```
1: $highest \leftarrow -\infty$
2: $lowest \leftarrow +\infty$
3: $tasks\_h \leftarrow$ tasks running on Host $h$
4: if $h$ has free resources then
5:     $lowest \leftarrow$ current time
6: else
7:     while $tasks\_h$ is not empty do
8:         $task \leftarrow$ next task on the list of tasks $tasks_h$
9:         $predicted\_end \leftarrow$ getStartTime($task$) + predictedTime($task$)
10:        if $predicted\_end$ < $lowest$ then
11:            $lowest \leftarrow predicted\_end$
12:        end if
13:    end while
14: end if
15: if $tasks\_h$ is empty then
16:     $highest \leftarrow$ current time
17: else
18:     while $tasks\_h$ is not empty do
19:         $task \leftarrow$ next task on the list of tasks $tasks_h$
20:         $predicted\_end \leftarrow$ getStartTime($task$) + predictedTime($task$)
21:         if $predicted\_end$ > $highest$ then
22:             $highest \leftarrow predicted_end$
23:         end if
24:     end while
25: end if
26: return $|highest - lowest|$
```

Finally, the algorithm for choosing which host will run a given task was divided into two for easier understanding. The algorithm 4 show the logic for selecting a host when there are available resources on the cluster, whereas the algorithm 6 is responsible for turning on a new host if it is justified.

The algorithm presented in 4 starts by checking if the number of tasks is lower or equal to the number of resources available, which is, as explained before, a rough approximation to determine if there are enough tasks to maintain the resources occupied (line 4). If this condition is verified, EASAHUM will try to mark hosts (line 5) following the algorithm 5, and then will calculate the best fit for that task (line 6) using the hosts that are available at the moment. If the best host was not marked or if its idle time is equal to or larger than the expected runtime of the task, it will use the best fit to run the task (lines 7-8). In the case that the best fit does not meet the conditions, EASAHUM will look for a host whose idle time is higher than the expected execution time of the task (lines 10-13). If there are more tasks than available hosts, and the condition presented in line (4) fails, EASAHUM will use the first available host (line 18), and unmark all hosts since there is no need to keep than marked anymore.

Algorithm 4 Choice of a machine for each task when there are available hosts

Input: t a task to schedule

Output: The host to execute the task t

```
1: $available\_hosts \leftarrow$ list of available hosts
2: $idle\_hosts \leftarrow$ list of idle hosts
3: if $idle\_hosts$ is not empty then
4:   if $getNumberRemainingTasks() \leq getNumberAvailableCores()$ then
5:     $MarkHosts(available\_hosts)$
6:     $best\_fit \leftarrow$ best fit for task $t$ on Host list $idle\_hosts$
7:     if $h$ is not marked or $predictTime(t) \leq getIdleTime(h)$ then
8:       return $best\_fit$
9:     end if
10:    while $idle\_hosts$ is not empty do
11:      $h \leftarrow$ next Host on the list of hosts $idle\_hosts$
12:      if $predictTime(t) \leq getIdleTime(h)$ then
13:        return $h$
14:      end if
15:    end while
16:  else
17:    Unmark all hosts
18:    return first Host of the list $idle\_hosts$
19:  end if
20: end if
```

When calculating the number of marked hosts, EASAHUM should not consider hosts that were turned off. This way, whenever a host is turned off, it will also be unmarked.

The process of marking the available hosts, presented on algorithm 5, will happen as long as the number of tasks is higher than the number of resources in unmarked hosts (line 1). So, while this condition is verified, EASAHUM will search among the list of hosts (lines 4-5) for the host with the lowest idle time that is not marked (lines 6-8). If any host was found, then it will be marked (lines 11-12).

Algorithm 5 Process of marking hosts

Input: $hosts$ a list of Hosts

```
1: while $getNumberRemainingTasks() \leq getNumberUnmarkedCores(hosts)$ and $hosts.size() - getNumberUnmarkedHosts(hosts) > 1$ do
2:     $min\_idle\_time \leftarrow +\infty$
3:     $min\_host \leftarrow None$
4:     while $hosts$ is not empty do
5:         $h \leftarrow$ next Host on the list of hosts $hosts$
6:         if $h$ is not marked and $getIdleTime(h) < min\_idle\_time$ then
7:             $min\_idle\_time \leftarrow getIdleTime(h)$
8:             $min\_host \leftarrow h$
9:         end if
10:    end while
11:    if $min\_host$ is not $None$ then
12:        mark $min\_host$
13:    end if
14: end while
```

The other part of the algorithm is presented in 6 and describes the behaviour of EASAHUM when there are no available resources. As explained previously, when the cluster has no resources to run a task and there are still a large number of tasks, EASAHUM will try to turn on a new host to run the pending tasks. It starts by checking if there is a host down (line 5) and if there is, it will turn it on (line 20) and use it (line 21). Otherwise, it will check if there is any host turned off to continue (line 6), and if there is not, it will not schedule the task (line 7). However, if there are available hosts turned off, it will start looking for one with the condition that if it is turned on, there must be more tasks than resources in the cluster (lines 9-11), since there is no need to turn on a new host just to run a few tasks, and so, if a host who meets this condition is found, EASAHUM will turn it on and create a new host on this new host (lines 12-14).

A special case happens at the start of the job execution or when all machines where turned off. In this case, the algorithm will have to turn on a new host even if there is only one task to schedule, which will correspond to the second part of the condition when searching for a new host to turn on (line 11). Due to this condition, whenever no hosts are turned on, the first host in the list of turned off hosts will be turned on.

Algorithm 6 Choice of a machine for each task when there are no available resources

Input: t a task to schedule

Output: The Host to execute the task h

```
 1: $available\_hosts \leftarrow$ list of available Hosts
 2: $underutilized\_hosts \leftarrow$ list of underutilized Hosts
 3: if $underutilized\_hosts$ is empty then
 4:     if no turned off hosts then
 5:         return $None$
 6:     end if
 7:     while list of turned off hosts is not empty do
 8:         $host \leftarrow$ next host on the list of turned off hosts
 9:         if number of remaining tasks $\geq$ number of cores on + $host.cores$ or no hosts turned on then
10:             turn on host $host$ return $host$
11:         end if
12:     end while
13:     return $None$
14: end if
```

Chapter 5

Implementation

5.1 Simulator

When deciding how to implement and test the algorithm, we have the option to either use a real environment and implement the algorithm in a tool, like Spark, or use a simulator. Using a real environment will give us more realistic and precise results when compared to those of a simulator, but will require much more resources to test and will be significantly harder to implement and analyse when compared to a simulator.

This way, we decided to use a simulator to implement and test the algorithm. To find a good simulator, we followed these criteria:

- The simulator must allow to turn the host on and off.
- The simulator must allow us to record the energy consumption of a workload execution.
- The simulator must allow an easy implementation of an algorithm.
- The simulator must have a tool to analyze the workload execution.
- Finally, the simulator must simulate an environment close to that of Spark.

The simulator we found that checks all these criteria was WRENCH [9], which is an open-source project that acts as a high-level interface to Simgrid [8], and has large support of packages like [3]. This simulator allows us to easily implement new algorithms and to run custom workloads, which will be helpful to test the jobs from HiBench previously executed on Spark, and also to calibrate the energy consumption of the environment in the simulator. Another great feature of this simulator is the existence of a dashboard that shows which host executed each task during the execution time, which will be a great tool for analyzing the performance of the algorithm since it allows us to visually see what machines are turned on and off at each time of the execution, and the level of utilization of each host.

Since it works as a high-level abstraction to Simgrid, it is very easy to use, in a way it only requires a workload file and a platform file description. The simulator also makes the task of creating a new algorithm very easy, since it only needs to sort the tasks that are ready to execute and then choose which machine should run them.

This simulator also allows changing the frequency of the simulated processor by defining a list of pstates in the platform file, where each pstate will correspond to a different level of performance represented in flops. If we define each pstate as the performance of the simulated hosts under a specific frequency, we can make the simulator use **DVFS** techniques to save energy by simply changing the current pstate of each processor. Later we will explain how the simulator calculates the energy consumption in section 5.2.

In the context of the simulator, each host may have multiple **VM**, and each **VM** can run a specific number of tasks up to the number of cores allocated to it. In order to make the environment as close to Spark as possible, we will make each machine have only one **VM**, making the same job as the Spark executor, with all cores allocated to it.

The simulator also allows us to suspend and shutdown **VMs** and hosts at any moment. Once again, we will simplify this by considering that we will always shutdown the **VM** alongside the host since it will only have a single **VM** at a time.

One of the main disadvantages of using a simulator is the energy consumption measuring since it will require an energy model, which is less accurate when compared to other options available in a real environment. However, the usage of an energy model still allows us to make comparisons with other algorithms despite not being fully accurate. Another disadvantage of the simulator is that the energy model only considers the energy consumed by the processor, disregarding other energy consumption sources like the network.

To test the simulator, we used traces of the workloads used previously in 3.2 and compared the execution time using the FIFO algorithm and a platform that represents the same one used to run the workloads that produced those traces. The results are presented in figure 27, which shows a graphical representation of those values. According to the results, the difference between the execution time on the simulator and other real scenarios is very similar. That small difference could be justified by using traces that were not produced by a test with those exact values.

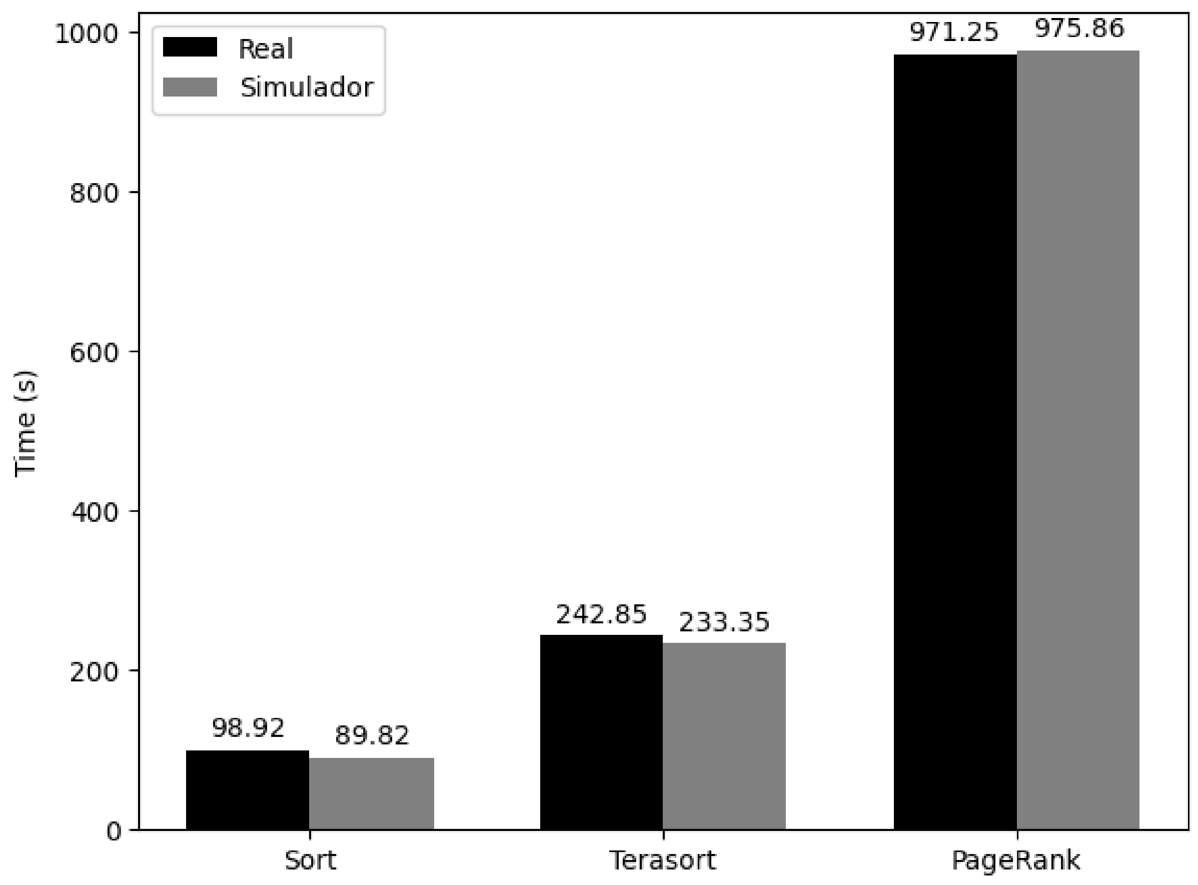


Figure 27: Execution time comparation

5.2 Energy model

Since a simulator does not rely on physical hosts, it needs an energy model to track the energy consumption of each host. As previously mentioned, WRENCH acts as an abstraction layer to Simgrid, and one of the many functionalities of Simgrid it supports is the host energy plugin [14]. Using an energy model, this plugin can measure the power the CPU consumes in each host at any time.

In order to calculate the power consumption of each machine, the model requires four values that must be specified for each host in the platform file describing the cluster used:

- Idle: the consumption when the host is turned on, but not running any task
- Epsilon: the consumption when all cores of the host are at 0%, but not in the Idle state
- AllCores: the consumption when all cores of the host are at 100% usage
- Off: the consumption when the host is turned off

Figure 28 shows these values and their influence on the model. As we can see, the power consumption of each machine can be calculated as in equation 5.1.

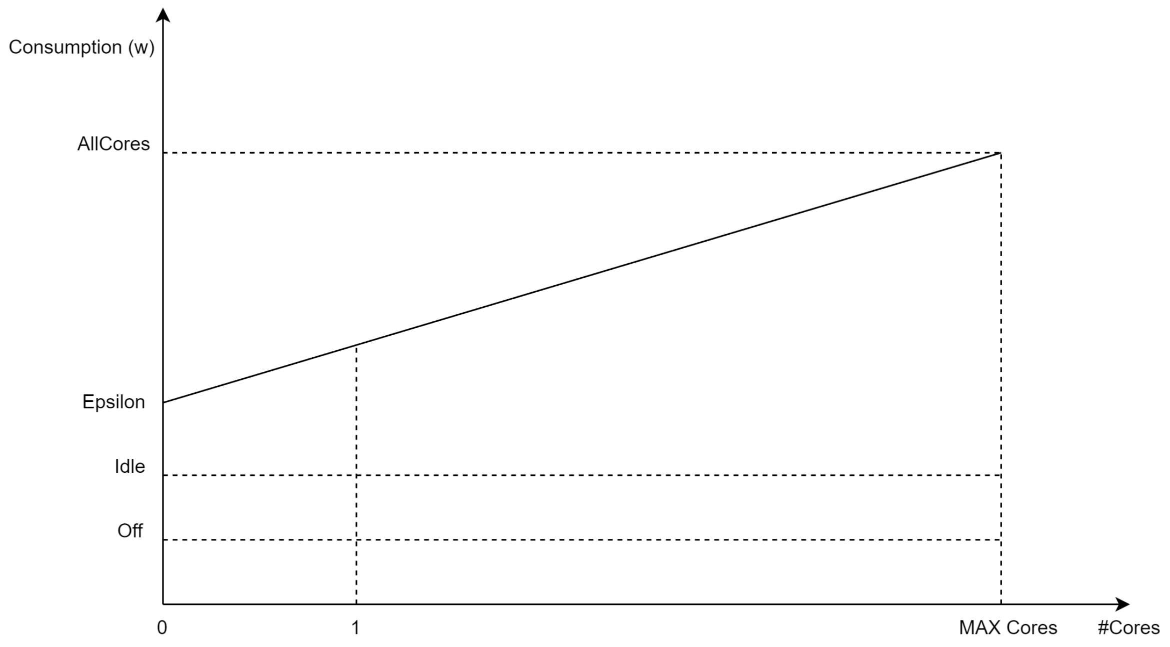


Figure 28: Energy Model

$$\begin{cases} \text{Power}_h(n) = \text{Off, if host } h \text{ is turned off} \\ \text{Power}_h(n) = \text{Idle, if host } h \text{ is idle} \\ \text{Power}_h(n) = \frac{\text{AllCores - Epsilon}}{\text{MAX Cores}} * n + \text{Epsilon, if host } h \text{ is running at least one task} \end{cases} \tag{5.1}$$

To find the best values for these parameters in our simulated platform, we ran a stress test on one of the machines for every number of available cores with the command *stress*, and recorded the power consumption of every number of cores using Powerjoular. The machine used has the same configuration as the ones used on previous tests. The results for this test are presented in table 6 and in figure 29.

Nrº de cores	0	1	2	3	4	5	6	7	8	9	10	11	12
Total (w)	0.5	18	28	38	48	55.8	63	63.8	65	67.3	69	70	71

Table 6: CPU stress test result

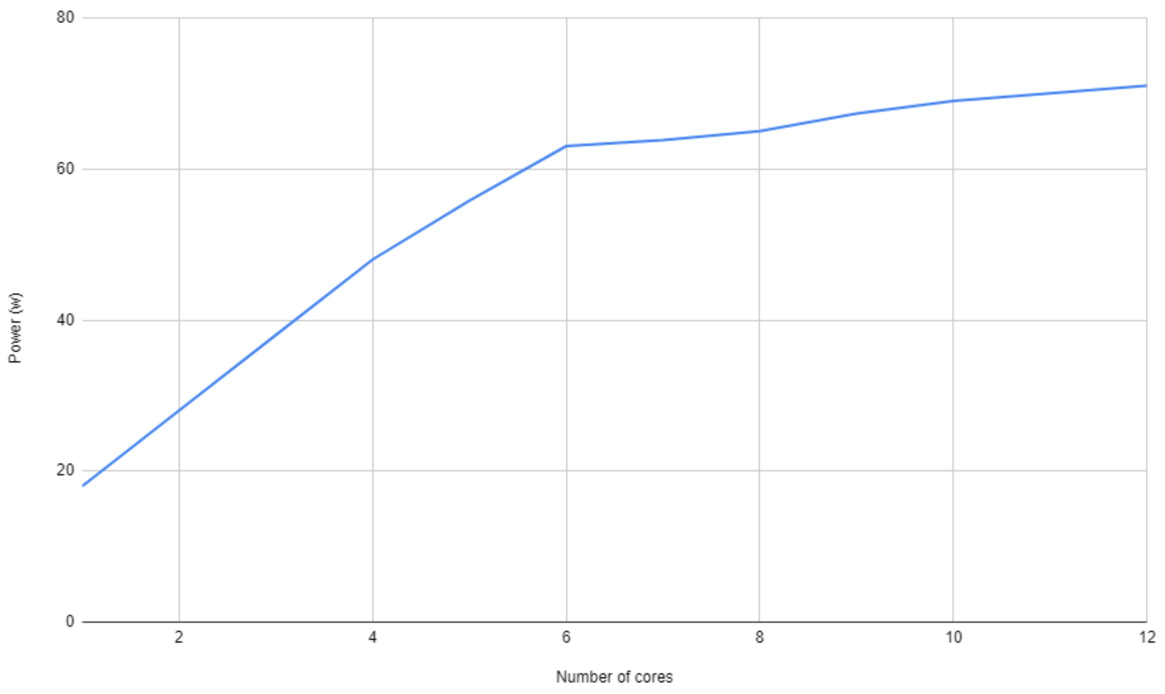


Figure 29: CPU stress test result graph

As we can see in those results, the power consumption increases linearly until six nodes, where it starts to increase very slightly until the twelve cores. This can be explained since the processor used has six physical cores, but supports up to twelve simultaneous threads. As we saw in section 3.6, running the workloads with more than six cores per executor can still decrease the execution time of the workload, so they should still count to the maximum number of cores simulated.

If we only consider the values of one and twelve cores in our model, we will end up with a very different result, especially for six cores, where we would have a difference of around 20 watts. So, in order to have a better approximation, we will increase the Epsilon to 40. This will also heavily penalize the usage of fewer cores, which follows the conclusions in section 3.6.

Regarding the other values, we will consider that a machined that is offline does not consume any energy, and when in an idle state, it consumes 5 watts. So Off will be 0 and Idle will be 5.

In order to confirm if the results produced by the model are comparable to the real measurements on the tests shown in section 3.2, we ran the traces of those executions on the simulator and recorded the results. The comparison between the power consumption on the simulator and the real scenario is presented in a graphical view 30. Contrarily to the execution time, we can see a difference in these results, which can be justified by the difficulty of making precise measurements by the simulator, due to differences in the tasks, where some may require more CPU than others, and even the difference shown between the real results of the stress test and the simulator. Although the model does not give us precise results, it provides a fair comparison and allows us to make conclusions.

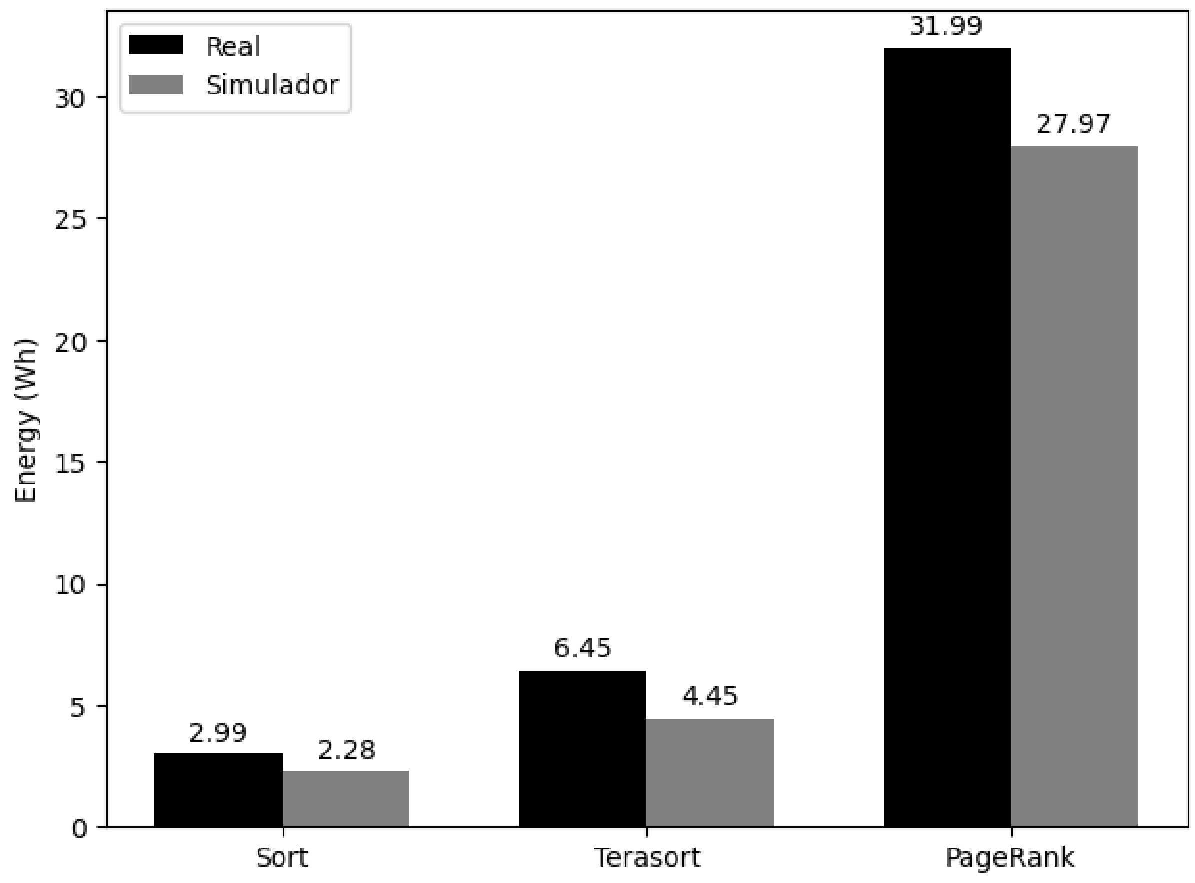


Figure 30: Energy consumption comparation

Although it will not be used in this work, Simgrid also supports power states (pstats), where we can specify various profiles for the CPU with different speeds and values for the power model. This allows for an easy use of **DVFS** techniques.

Even though the CPU is the main source of energy consumption in a machine, there are also other sources like the network. However, the simulator used in this work does not support by default ways of measuring the power consumption from the network links. Moreover, we do not have a software tool that can measure the power consumption of the links, as we have for the CPU, which makes recording this type of energy consumption harder in our real scenario.

Chapter 6

Evaluation

6.1 Testing methodology

After implementing the algorithm in the simulator, we ran a set of tests in order to evaluate its performance using traces of executions. Firstly, we used traces from the same workloads as in section 3.2, with the following configurations:

- Sort with a datasize of 55 GB
- Terasort a datasize of 500 MB
- Pagerank with 8000000 pages, 3 iterations and 16 block_width

In addition to those workloads, we also created a new workload formed by the combination of all the previous workloads, which we called All-in-one. This workload will be important to test how the algorithm works in a huge quantity of tasks very different from one another. This type of heterogeneous workload is a common occurrence on clusters shared by multiple users, so this workload will also serve to evaluate the performance of EASAHUM in this specific scenario.

We also added a final synthetic workload 1000genome-chameleon-20ch-250k, which we will call Genome for simplification and is available in [3]. This workload will allow us to test how the algorithm works with a workload composed of a huge quantity of long tasks, and also see how the algorithm reacts to a workload that was not generated by a Spark execution.

During this test, we will evaluate the following parameters:

- Power consumption
- Execution time
- Time spent in the algorithm

The power consumption will represent the total power consumed by the simulated platform according to the energy model discussed in chapter 5.2. This will be our main parameter of comparison since it is the main objective of the algorithm to reduce energy consumption. As a secondary cooperation, we will also analyze the execution time of the workload to verify that the algorithm does not sacrifice the execution time in favour of energy consumption. Finally, we will also compare the time spent in the algorithm calculation to see if it is negligible when compared to the execution time of the workload.

To compare the performance of EASAHUM, we will use E-FIFO, the energy-aware version of FIFO. This algorithm was chosen because it has a similar technique for shutting down hosts, where the hosts are turned off whenever they are not running any task, and also for being a straightforward algorithm, which is a good comparison for the time spent on algorithm calculations. Since EASAHUM uses a similar logic to FIFO when there are lots of tasks to schedule, we can also evaluate the impact of increasing the host utilization on the energy consumption when there are few tasks ready to schedule.

Since we are running these tests on a simulator, we know exactly the execution time of each task. However, we want to test how the accuracy of the execution time prevision of the tasks affects results. So, when EASAHUM tries to predict the execution time, we will use a value that is around the real value with a certain percentage of uncertainty. This way, we will run the test for a set of accuracy values for the case of EASAHUM, and analyse how this affects the results. In the case of E-FIFO, we will use the same results since the value of the uncertainty will not affect the scheduling process.

We are expecting All-in-one to have the best energy consumption results since the algorithm can take advantage of the different execution times of the different tasks. Also, we expect the time spent on calculations to be higher on EASAHUM since the operation to sort tasks is expensive and is not present on E-FIFO.

Finally, the platform used for this test is composed of three workers that represent the same three workers used during the previous tests, and have the same parameters for the energy models as the ones used in 5.2.

6.2 Results

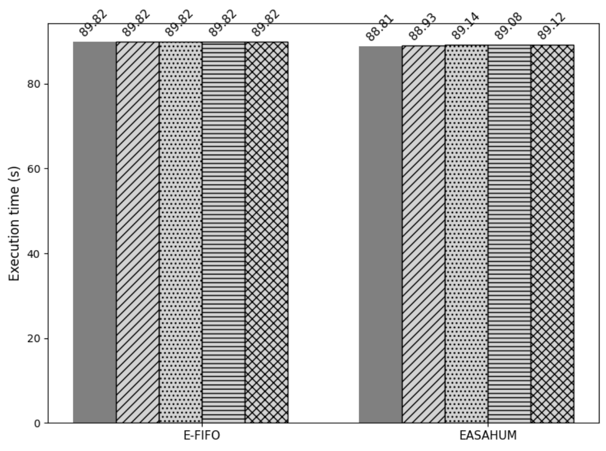
The results for the comparison of the execution time between E-FIFO and EASAHUM across all levels of uncertainty are presented in figure 31. On the Sort 31a and Terasort 31b the execution time of EASAHUM is very similar to the one on E-FIFO, which is to be expected since those workloads are very simple and so, E-FIFO is already able to achieve an almost perfect scheduling strategy, and both workloads, the execution time degraded slightly with the increase of the uncertainty in predicting the tasks execution time.

In the case of Genome 31d, there was a small increase of around 4% in the execution time, which can be explained by one task that ran very late at the execution and is a dependency for a set of tasks that run at the end of the job, which in turn increases the underutilization at the end of the execution, as we can see in figure 33. Although the execution produced by E-FIFO had some problems where it did not fully utilize its resources, as we can see in figure 32, it was still slightly faster than our algorithm.

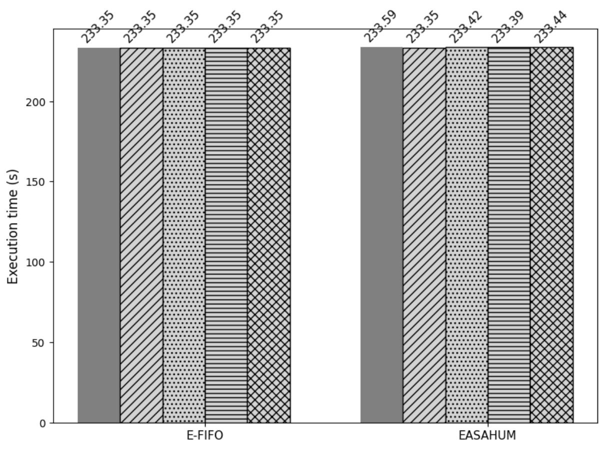
The case of Pagerank for the execution time 31c is very interesting since it was almost 19% slower with EASAHUM with 0% uncertainty in comparison to E-FIFO, but as we will see when analyzing the results for the energy consumption, it was able to reduce the total energy consumed by 3.7%. This decrease in the execution time is due to the usage of only two workers instead of three.

There is another very interesting case in All-in-one 31e. For 0% uncertainty, EASAHUM was able to reduce the execution time by more than 12% in comparison to E-FIFO. This happens because EASAHUM can schedule tasks in a way that takes advantage of the differences in execution time across all different tasks. However, EASAHUM was also heavily penalized with the increase in the degree of uncertainty, becoming slower than the execution on E-FIFO when the uncertainty was 10%. However, it reduced when the uncertainty increased to 15%, which can be explained by an occasional better placement of tasks by EASAHUM on these specific cases due to the uncertainty, and should not be seen as an improvement, since the rest of the tasks did not have relevant changes.

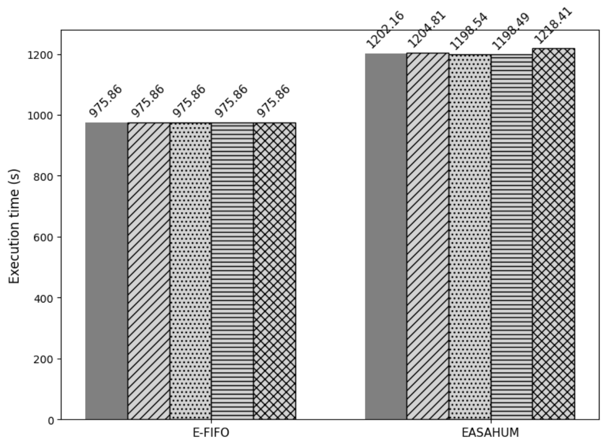
With these results, we can see that in most cases the execution result of EASAHUM will not be slower than the one from E-FIFO, and in the cases where we have a heterogeneous load, it is possible to achieve a better execution time.



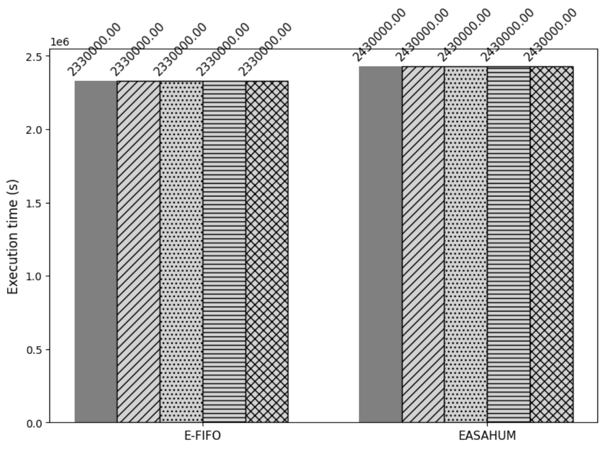
(a) Execution time on Sort



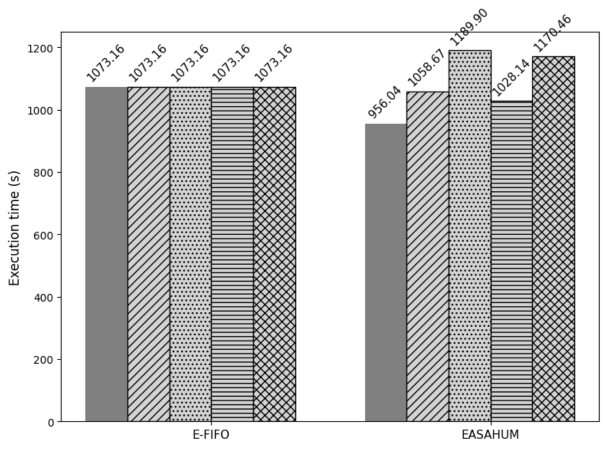
(b) Execution time on Terasort



(c) Execution time on Pagerank



(d) Execution time on Genome



(e) Execution time on All-in-one

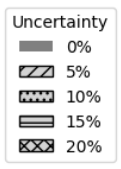


Figure 31: E-FIFO vs EASAHUM execution time

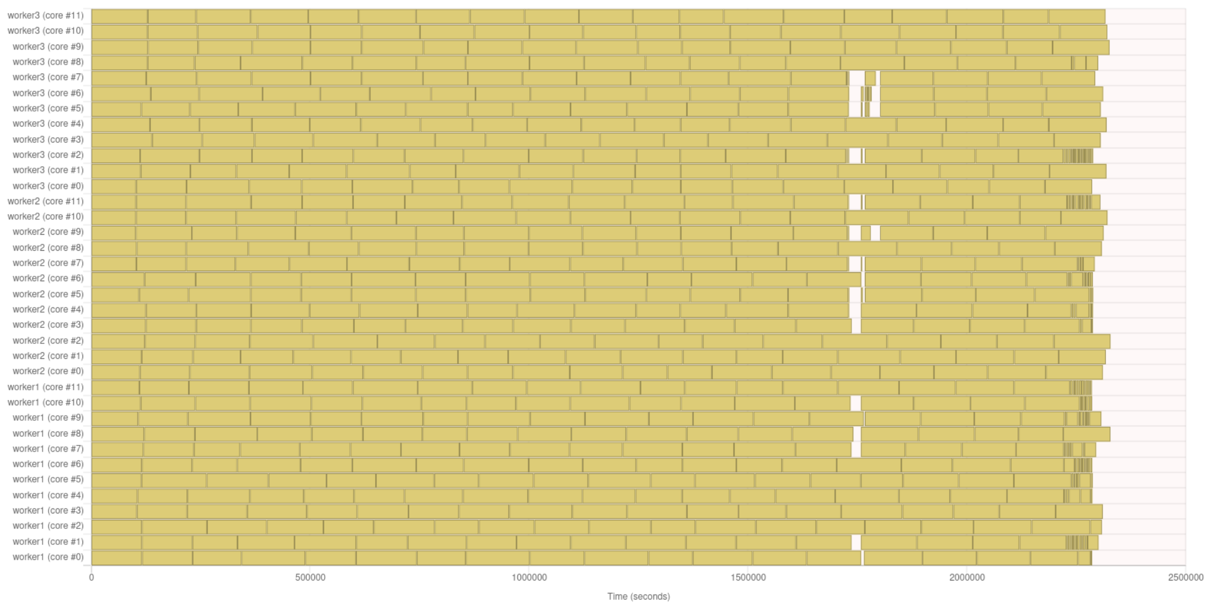


Figure 32: Execution of Genome with E-FIFO and 0% uncertainty

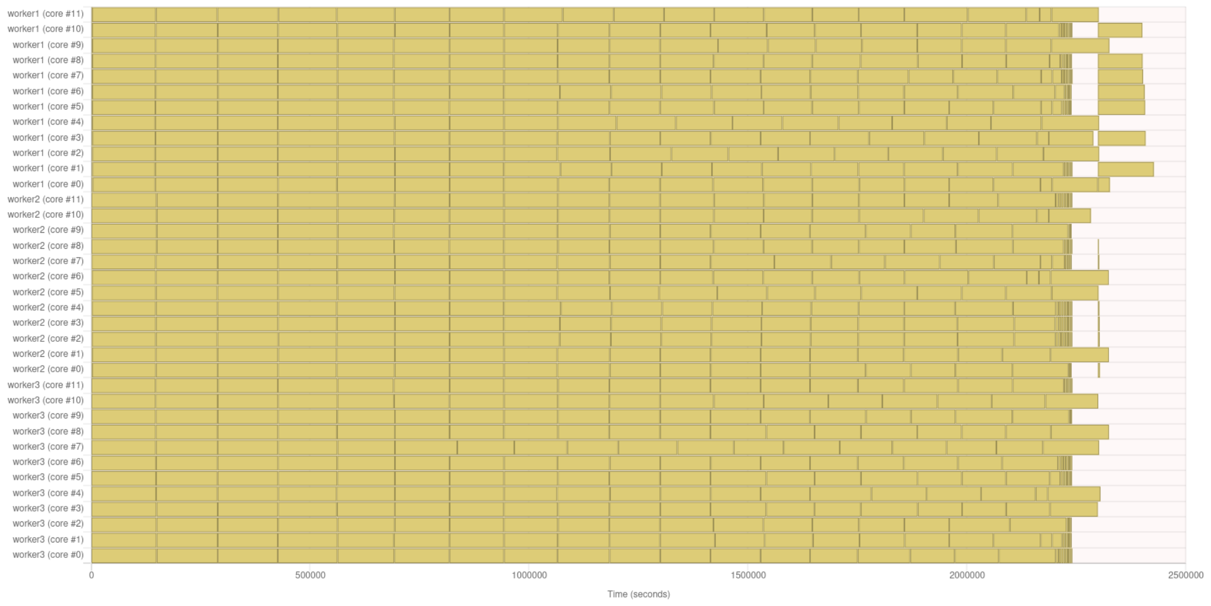


Figure 33: Execution of Genome with EASAHUM and 0% uncertainty

Going into the results of comparing the two algorithms regarding energy consumption in figure 34, we can see that when we have no uncertainty regarding the execution time of the tasks, we can achieve better energy results on all workloads except for Genome. The explanation for the slight increase in power consumption for the Genome workload on EASAHUM is the same as the one for its execution time, where the increase in execution time due to the wrong ordering of tasks also increases the execution time and the host underutilization time. However, the increase in power consumption was much smaller than the increase in execution time, since it was able to make good scheduling decisions during the rest of the job.

In the case of both Sort 31a and Terasort 31b, although the results for the energy consumption were slightly better for 0% uncertainty, they become practically equal to the ones of E-FIFO with the increase of the uncertainty. However, even at 20% uncertainty, it was not worse than E-FIFO.

Although the execution time of Pagerank was higher in EASAHUM when compared to E-FIFO 31b, the power consumption was about 3.7% lower on EASAHUM 34c. This decrease in power consumption is due to the usage of only two workers instead of three. The reason why the saving in energy was not higher is because the usage of only two hosts also increased the execution time, which makes the cluster consume less energy but need much more time to execute the workload. This result is a good example of how the increase in execution time does not represent a direct increase in energy consumption.

Finally, All-in-one is again an interesting case 34c. This workload was the one that showed the best results for energy consumption, consuming less than 16% of energy in EASAHUM in comparison to E-FIFO for 0% uncertainty, and even more than 6% less energy for 20% uncertainty. So, it was able to save energy even when the degree of uncertainty was very high, and just like on Pagerank, EASAHUM was able to save energy even when the execution time was higher.

The All-in-one workload is a good example of how EASAHUM achieves energy savings. Figure 35 shows the execution of All-in-one with the E-FIFO algorithm. As we can see, there are several moments where the hosts are executing only two or three tasks, and most of them finish at different times. As we can see in figure 36, EASAHUM can schedule the tasks in a way that the hosts are not underutilized during long periods, resulting in a reduction of energy consumption, and also a reduction in execution time.

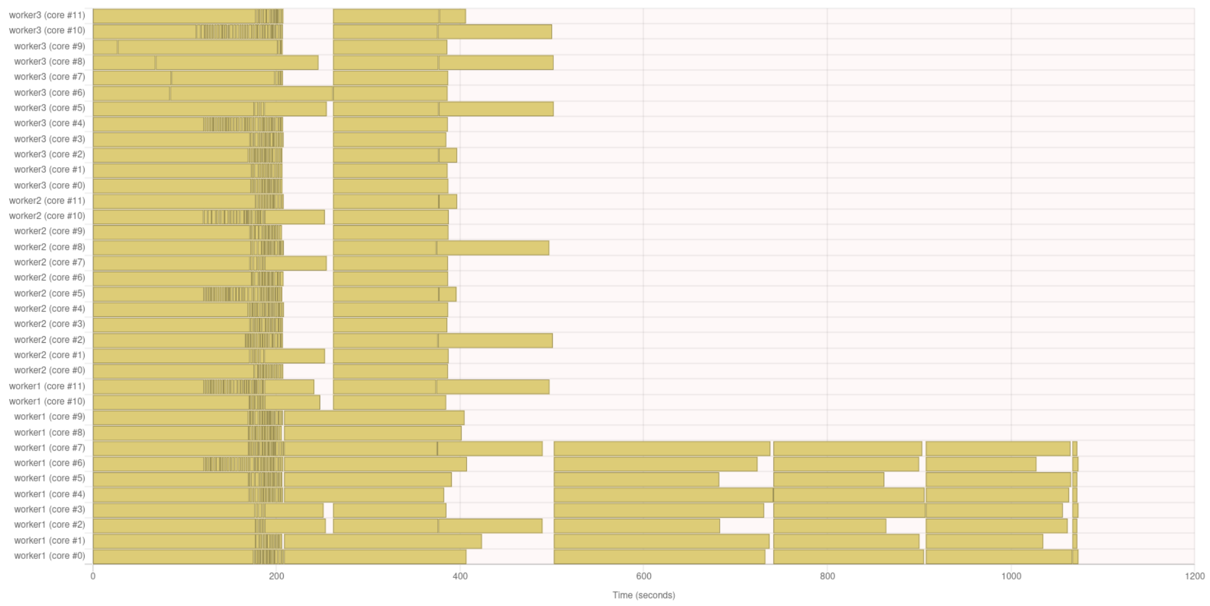
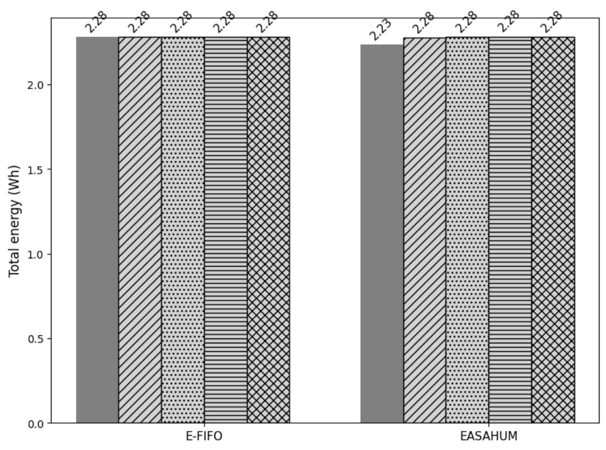
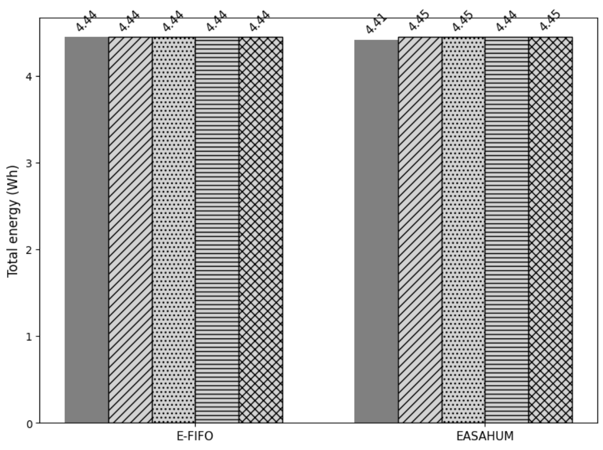


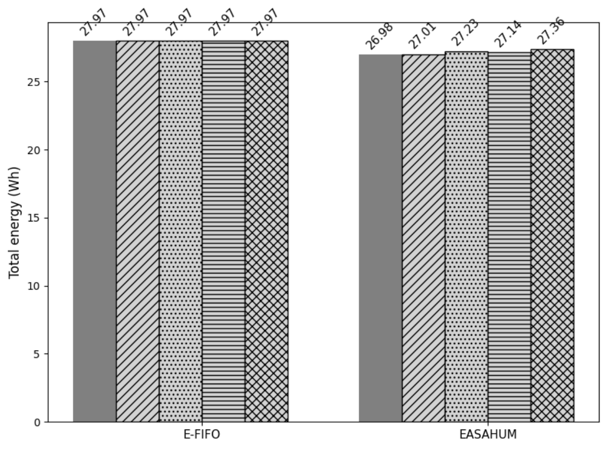
Figure 35: Execution of All-in-one with E-FIFO and 0% uncertainty



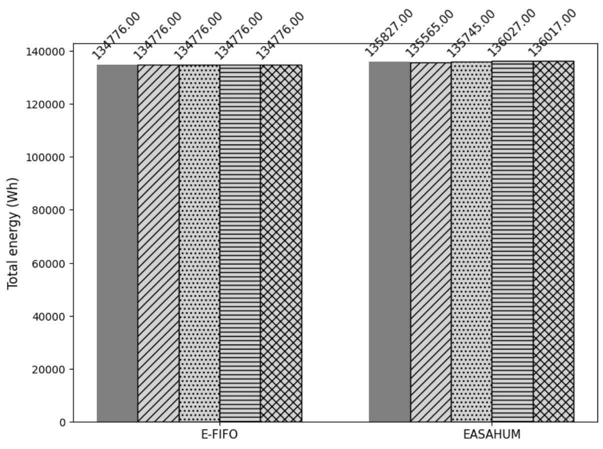
(a) Power consumption on Sort



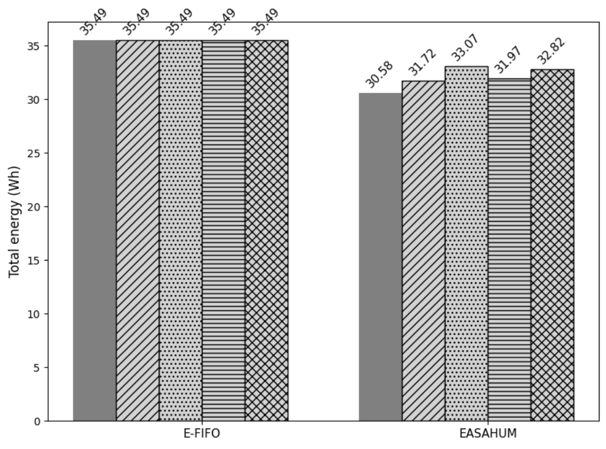
(b) Power consumption on Terasort



(c) Power consumption on Pagerank



(d) Power consumption on Genome



(e) Power consumption on All-in-one

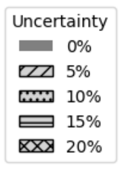


Figure 34: E-FIFO vs EASAHUM power consumption

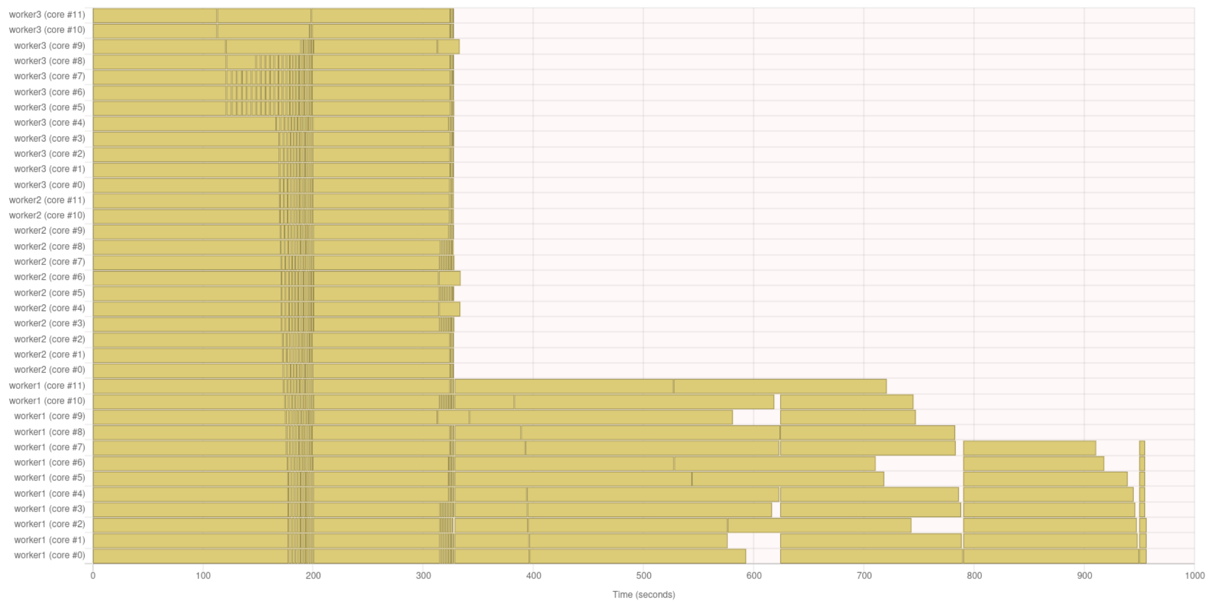
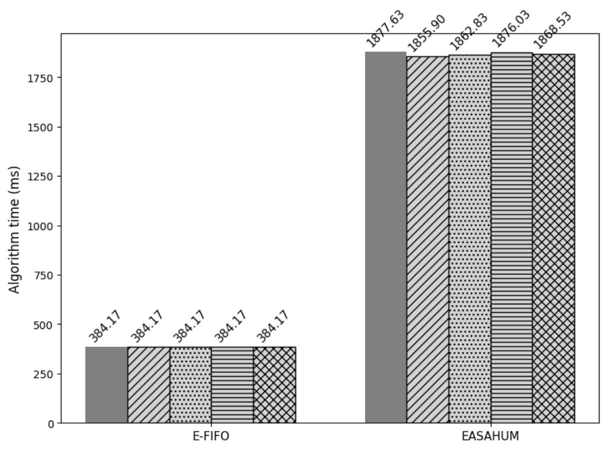
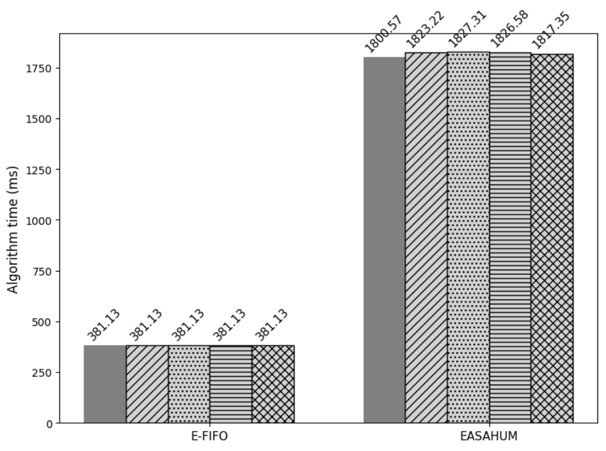


Figure 36: Execution of All-in-one with EASAHUM and 0% uncertainty

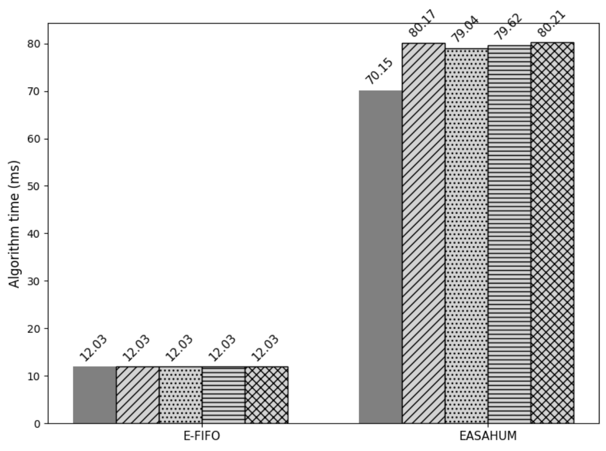
Finally, the results of the comparison of the time spent on the calculation of both algorithms can be seen in 37. We can see that across all workloads, the results for EASAHUM are worse than the ones for E-FIFO, which is to be expected since EASAHUM is a more complex algorithm. The worst results are on All-in-one with 20% uncertainty 37e, in which EASAHUM spent about 85% more time than E-FIFO on calculation, and the best results happen in Genome 37d with 10% uncertainty, where EASAHUM spent more 69.35% more time than E-FIFO. However, despite this increase, the time spent on calculations for the algorithm was still just a fraction of the execution time. These results also show that the increase in the level of uncertainty does not affect the complexity of the algorithm, since it remained approximately the same across all levels of uncertainty.



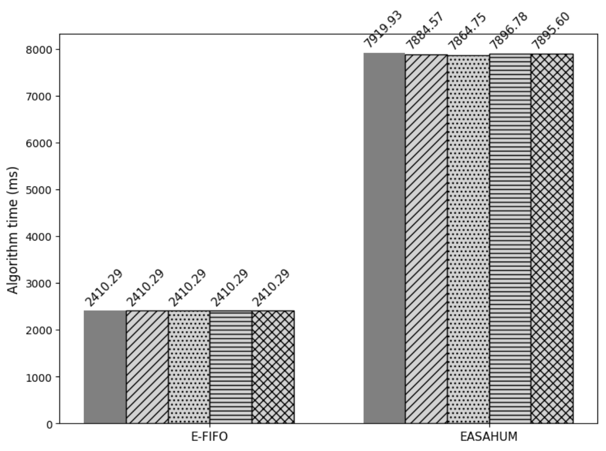
(a) Scheduling time on Sort



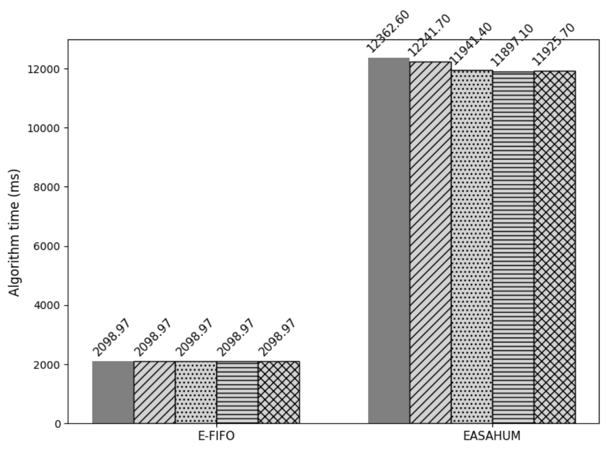
(b) Scheduling time on Terasort



(c) Scheduling time on Pagerank



(d) Scheduling time on Genome



(e) Scheduling time on All-in-one

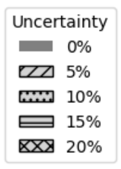


Figure 37: E-FIFO vs EASAHUM scheduling time

Chapter 7

Conclusion and future work

Over recent years, many scheduling algorithms have been proposed for data processing tools to reduce energy efficiency, and most of those algorithms are developed to work with heterogeneous environments to take advantage of the differences among all the different configurations. However, data centers are typically homogeneous clusters, so they can not use these algorithms to their full potential.

The analysis of Spark workloads with FIFO and FAIR algorithms, which are implemented by default in Spark, shows that there is no significant difference between them regarding energy consumption. Further analysis revealed that energy consumption can be decreased by increasing the number of tasks that run on each Spark executor, as long as the processor is not running on maximum frequency.

In this dissertation we propose EASAHUM, a new scheduling algorithm capable of reducing the energy consumption in homogeneous environments, without compromising the execution time and also not increasing the scheduling time complexity. To test EASAHUM, we used traces of real execution from a Spark cluster and synthetic workloads, which proved that EASAHUM can effectively reduce energy consumption up to 16%, and even reduce the execution time up to 12.25%, specifically when the workload is composed of tasks with many different execution times.

As for future work, we would like to implement DVFS techniques to take advantage of the way EASAHUM keeps the hosts with a high usage percentage and also because it can reduce execution time. These two characteristics make it a good candidate for reducing the processor frequency to save even more power. Moreover, it would be relevant to analyze the power consumption effects in other parts of the cluster, such as the network. Finally, we can test if increasing the complexity in certain parts of the algorithm, such as the detection of when there are not enough tasks in the cluster to keep the cluster with high utilization, can reduce the energy consumption even more without compromising the execution time.

Bibliography

[1] November 2022 | TOP500. URL https://www.top500.org/lists/green500/2022/11/.

[2] Job Scheduling - Spark 3.3.1 Documentation. URL https://spark.apache.org/docs/latest/job-scheduling.html.

[3] Pegasus Workflow Execution Instances, May 2023. URL https://github.com/wfcommons/pegasus-instances. original-date: 2016-11-27T05:14:05Z.

[4] Thomas Anderson, Adam Belay, Mosharaf Chowdhury, Asaf Cidon, and Irene Zhang. Treehouse: A Case For Carbon-Aware Datacenter Software, January 2022. URL http://arxiv.org/abs/2201.02120. arXiv:2201.02120 [cs].

[5] Josep Ll. Berral, Íñigo Goiri, Ramón Nou, Ferran Julià, Jordi Guitart, Ricard Gavaldà, and Jordi Torres. Towards energy-aware scheduling in data centers using machine learning. In *Proceedings of the 1st International Conference on Energy-Efficient Computing and Networking*, e-Energy '10, pages 215–224, New York, NY, USA, April 2010. Association for Computing Machinery. ISBN 978-1-4503-0042-1. doi: 10.1145/1791314.1791349. URL https://doi.org/10.1145/1791314.1791349.

[6] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache flink™: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4):28–38, 2015.

[7] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache flink™: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4):28–38, 2015.

[8] H. Casanova. Simgrid: a toolkit for the simulation of application scheduling. In *Proceedings First IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 430–437, 2001. doi: 10.1109/CCGRID.2001.923223.

[9] Henri Casanova, Rafael Ferreira da Silva, Ryan Tanaka, Suraj Pandey, Gautam Jethwani, William Koch, Spencer Albrecht, James Oeth, and Frederic Suter. Developing Accurate and Scalable Simulators of Production Workflow Management Systems with WRENCH, November 2020. URL `https://github.com/wrench-project/wrench`.

[10] Pawel Czarnul, Jerzy Proficz, and Adam Krzywaniak. Energy-Aware High-Performance Computing: Survey of State-of-the-Art Tools, Techniques, and Environments. *Scientific Programming*, 2019:e8348791, April 2019. ISSN 1058-9244. doi: 10.1155/2019/8348791. URL `https://www.hindawi.com/journals/sp/2019/8348791/`. Publisher: Hindawi.

[11] Nathan C. Frey, Baolin Li, Joseph McDonald, Dan Zhao, Michael Jones, David Bestor, Devesh Tiwari, Vijay Gadepally, and Siddharth Samsi. Benchmarking Resource Usage for Efficient Distributed Deep Learning, January 2022. URL `http://arxiv.org/abs/2201.12423`. arXiv:2201.12423 [cs].

[12] Rong Ge, Xizhou Feng, and K.W. Cameron. Performance-constrained Distributed DVS Scheduling for Scientific Applications on Power-aware Clusters. pages 34–34, December 2005. ISBN 978-1-59593-061-3. doi: 10.1109/SC.2005.57.

[13] Jatinder N. D. Gupta and Johnny C. Ho. A new heuristic algorithm for the one-dimensional bin-packing problem. *Production Planning & Control*, 10(6):598–603, January 1999. ISSN 0953-7287. doi: 10.1080/095372899232894. URL `https://doi.org/10.1080/095372899232894`. Publisher: Taylor & Francis _eprint: https://doi.org/10.1080/095372899232894.

[14] Franz Christian Heinrich, Tom Cornebize, Augustin Degomme, Arnaud Legrand, Alexandra Carpen-Amarie, Sascha Hunold, Anne-Cécile Orgerie, and Martin Quinson. Predicting the energy-consumption of mpi applications at scale using only a single node. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 92–102, 2017. doi: 10.1109/CLUSTER.2017.66.

[15] Shengsheng Huang, Jie Huang, Jinquan Dai, Tao Xie, and Bo Huang. The HiBench benchmark suite: Characterization of the MapReduce-based data analysis. In *2010 IEEE 26th International Conference on Data Engineering Workshops (ICDEW 2010)*, pages 41–51, March 2010. doi: 10.1109/ICDEW.2010.5452747.

[16] Muhammed Tawfiqul Islam, Huaming Wu, Shanika Karunasekera, and Rajkumar Buyya. SLA-Based Scheduling of Spark Jobs in Hybrid Cloud Computing Environments. *IEEE Transactions on Computers*, 71(5):1117–1132, May 2022. ISSN 1557-9956. doi: 10.1109/TC.2021.3075625. Conference Name: IEEE Transactions on Computers.

[17] Tang Jianchao, Yang Shuqiang, Huang Chaoqiang, and Yan Zhou. Design and Implementation of Scheduling Pool Scheduling Algorithm Based on Reuse of Jobs in Spark. In *2016 IEEE First International Conference on Data Science in Cyberspace (DSC)*, pages 290–295, June 2016. doi: 10.1109/DSC.2016.81.

[18] Nicola Jones. How to stop data centres from gobbling up the world's electricity, Sep 2018. URL https://www.nature.com/articles/d41586-018-06610-y#ref-CR1.

[19] Etienne Le Sueur and Gernot Heiser. Dynamic voltage and frequency scaling: The laws of diminishing returns. In *Proceedings of the 2010 International Conference on Power Aware Computing and Systems*, HotPower'10, page 1–8, USA, 2010. USENIX Association.

[20] Hongjian Li, Huochen Wang, Shuyong Fang, Yang Zou, and Wenhong Tian. An energy-aware scheduling algorithm for big data applications in Spark. *Cluster Computing*, 23(2):593–609, June 2020. ISSN 1573-7543. doi: 10.1007/s10586-019-02947-9. URL https://doi.org/10.1007/s10586-019-02947-9.

[21] Hongjian Li, Yaojun Wei, Yu Xiong, Enjie Ma, and Wenhong Tian. A frequency-aware and energy-saving strategy based on dvfs for spark. *The Journal of Supercomputing*, 77, 10 2021. doi: 10.1007/s11227-021-03740-5.

[22] Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. Learning scheduling algorithms for data processing clusters. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, pages 270–288, New York, NY, USA, August 2019. Association for Computing Machinery. ISBN 978-1-4503-5956-6. doi: 10.1145/3341302.3342080. URL https://doi.org/10.1145/3341302.3342080.

[23] Olli Mämmelä, Mikko Majanen, Robert Basmadjian, Hermann Meer, André Giesler, and Willi Homberg. Energy-aware job scheduler for high-performance computing. *Computer Science - Research and Development*, 27, 11 2012. doi: 10.1007/s00450-011-0189-6.

[24] Dawn Nafus, Eve M. Schooler, and Karly Ann Burch. Carbon-Responsive Computing: Changing the Nexus between Energy and Computing. *Energies*, 14(21):6917, January 2021. ISSN 1996-1073. doi: 10.3390/en14216917. URL https://www.mdpi.com/1996-1073/14/21/6917. Number: 21 Publisher: Multidisciplinary Digital Publishing Institute.

[25] Mitra Nasri, Robert I Davis, and Bjorn B Brandenburg. Open Problems in FIFO Scheduling with Multiple Offsets.

[26] Adel Noureddine. PowerJoular and JoularJX: Multi-Platform Software Power Monitoring Tools. In *2022 18th International Conference on Intelligent Environments (IE)*, pages 1–4, Biarritz, France, June 2022. IEEE. ISBN 978-1-66546-934-0. doi: 10.1109/IE54923.2022.9826760. URL https://ieeexplore.ieee.org/document/9826760/.

[27] Thanh-Phuong Pham, Juan J. Durillo, and Thomas Fahringer. Predicting workflow task execution time in the cloud using a two-stage machine learning approach. *IEEE Transactions on Cloud Computing*, 8(1):256–268, 2020. doi: 10.1109/TCC.2017.2732344.

[28] Gustavo Pinto and Fernando Castor. Energy efficiency: A new concern for application software developers. *Commun. ACM*, 60(12):68–75, nov 2017. ISSN 0001-0782. doi: 10.1145/3154384. URL https://doi.org/10.1145/3154384.

[29] V. K. Mohan Raj and R. Shriram. Power management in virtualized datacenter – A survey. *Journal of Network and Computer Applications*, 69:117–133, July 2016. ISSN 1084-8045. doi: 10.1016/j.jnca.2016.04.019. URL https://www.sciencedirect.com/science/article/pii/S1084804516300704.

[30] Yogesh Sharma, Bahman Javadi, Weisheng Si, and Daniel Sun. Reliability and energy efficiency in cloud computing systems: Survey and taxonomy. *Journal of Network and Computer Applications*, 74:66–85, October 2016. ISSN 1084-8045. doi: 10.1016/j.jnca.2016.08.010. URL https://www.sciencedirect.com/science/article/pii/S1084804516301746.

[31] Wenhu Shi, Hongjian Li, Junzhe Guan, Hang Zeng, and Rafe Misskat jahan. Energy-efficient scheduling algorithms based on task clustering in heterogeneous spark clusters. *Parallel Computing*, 112:102947, September 2022. ISSN 0167-8191. doi: 10.1016/j.parco.2022.102947. URL https://www.sciencedirect.com/science/article/pii/S0167819122000436.

[32] Wenhu Shi, Hongjian Li, and Hang Zeng. DRL-based and Bsld-Aware Job Scheduling for Apache Spark Cluster in Hybrid Cloud Computing Environments. *Journal of Grid Computing*, 20(4):44, December 2022. ISSN 1570-7873, 1572-9184. doi: 10.1007/s10723-022-09630-1. URL https://link.springer.com/10.1007/s10723-022-09630-1.

[33] Zhuo Tang, Ling Qi, Zhenzhen Cheng, Kenli Li, Samee Khan, and Keqin Li. An Energy-Efficient Task Scheduling Algorithm in DVFS-enabled Cloud Environment. *Journal of Grid Computing*, 14, April 2015. doi: 10.1007/s10723-015-9334-y.

[34] H. Topcuoglu, S. Hariri, and Min-You Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Trans. Parallel Distrib. Syst.*, 13(3):260–274, March 2002. ISSN 10459219. doi: 10.1109/71.993206. URL http://ieeexplore.ieee.org/document/993206/.

[35] J. D. Ullman. NP-complete scheduling problems. *Journal of Computer and System Sciences*, 10(3): 384–393, June 1975. ISSN 0022-0000. doi: 10.1016/S0022-0000(75)80008-0. URL https://www.sciencedirect.com/science/article/pii/S0022000075800080.

[36] Vasanth Venkatachalam and Michael Franz. Power reduction techniques for microprocessor systems. *ACM Computing Surveys*, 37(3):195–237, September 2005. ISSN 0360-0300. doi: 10.1145/1108956.1108957. URL https://doi.org/10.1145/1108956.1108957.

[37] Lizhe Wang, Jie Tao, Gregor von Laszewski, and Dan Chen. Power Aware Scheduling for Parallel Tasks via Task Clustering. In *2010 IEEE 16th International Conference on Parallel and Distributed Systems*, pages 629–634, December 2010. doi: 10.1109/ICPADS.2010.128. ISSN: 1521-9097.

[38] Lizhe Wang, Samee U. Khan, Dan Chen, Joanna Kołodziej, Rajiv Ranjan, Cheng-zhong Xu, and Albert Zomaya. Energy-aware parallel task scheduling in a cluster. *Future Generation Computer Systems*, 29(7):1661–1670, September 2013. ISSN 0167739X. doi: 10.1016/j.future.2013.02.010. URL https://linkinghub.elsevier.com/retrieve/pii/S0167739X13000484.

[39] Marilyn Wolf. Chapter 5 - processors and systems. In Marilyn Wolf, editor, *The Physics of Computing*, pages 149–203. Morgan Kaufmann, Boston, 2017. ISBN 978-0-12-809381-8. doi: https://doi.org/10.1016/B978-0-12-809381-8.00005-5. URL https://www.sciencedirect.com/science/article/pii/B9780128093818000055.

[40] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing.

[41] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez,

Scott Shenker, and Ion Stoica. Apache Spark: a unified engine for big data processing. *Communications of the ACM*, 59(11):56–65, October 2016. ISSN 0001-0782. doi: 10.1145/2934664. URL https://doi.org/10.1145/2934664.

[42] Shuai Zhao. A FIFO Spin-based Resource Control Framework for Symmetric Multiprocessing. URL https://core.ac.uk/reader/160470220.

Place here information about funding, FCT project, etc. in which the work is framed. Leave empty otherwise.