Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE

Corso di Laurea Magistrale in Informatica

DEVELOPMENT OF DATA-DRIVEN DISPATCHING HEURISTICS FOR HETEROGENEOUS HPC SYSTEMS

Relatore:
Chiar.mo Prof.
Ozalp Babaoglu

Presentata da:
Alessio Netti

Correlatore:
Chiar.ma Prof.ssa
Zeynep Kiziltan

II Sessione
Anno Accademico 2016/2017

"When we first went out into space, you showed me the Galaxy. Do you remember?"

"Of course."

"You speeded time and the Galaxy rotated visibly. And I said, as though anticipating this very time, 'The Galaxy looks like a living thing, crawling through space.' Do you think that, in a way, it is alive already?"

[Isaac Asimov - Foundation's Edge]

Introduzione

Nell'ambito dei sistemi *High-Performance Computing*, realizzare euristiche di *dispatching* efficaci è fondamentale al fine di ottenere buoni livelli di *Quality of Service*. Per dispatching intendiamo i metodi tramite cui i task (o *jobs*) sottomessi dagli utenti al sistema sono selezionati e preparati per l'avvio su di esso, sia in termini *temporali* che di *allocazione* delle risorse. In questo contesto, ci concentreremo sul design e l'analisi di **euristiche di allocazione** per il dispatching; tali euristiche saranno progettate per sistemi HPC *eterogenei*, nei quali i vari nodi possono essere equipaggiati con diverse tipologie di unità di elaborazione. Alcune di esse, inoltre, saranno di tipo **data-driven**, e dunque sfrutteranno l'informazione fornita dal *workload* corrente in modo da stimare parametri ignoti del sistema, e migliorare la propria efficacia.
Considereremo in particolare **Eurora**, un sistema HPC eterogeneo realizzato da CINECA, a Bologna, oltre che un workload catturato dal relativo log di sistema, contenente jobs reali inviati dagli utenti. Un contesto di tal genere, in piccola scala ed eterogeneo, costituisce l'ambiente perfetto per la valutazione di diversi metodi di dispatching. Tutto ciò è stato possibile grazie ad **AccaSim**, un simulatore di sistemi HPC da noi sviluppato nel Dipartimento di Informatica - Scienza e Ingegneria (DISI) dell'Università di Bologna: AccaSim è uno strumento innovativo per l'analisi dei sistemi HPC, il quale ha attualmente pochissimi rivali in termini di flessibilità ed efficienza.

In particolare, quest'elaborato affronta il tema della valutazione di metodi di dispatching HPC in un ambiente simulato, insieme all'impiego di euristiche data-driven per la predizione della durata dei jobs. Ciò è stato fatto al fine di stimare l'impatto di tali tecniche sul *throughput* del sistema, in termini di tempi di attesa e dimensione della coda dei jobs, ancora una volta nell'ambito dei sistemi HPC eterogenei, più difficili da gestire rispetto alle controparti omogenee.
Il contributo principale di questo lavoro consiste nel design e nello sviluppo di nuove euristiche di allocazione: queste sono state impiegate insieme a metodi

di scheduling già disponibili, i quali sono stati a loro volta adattati e migliorati. Le euristiche sviluppate sono state poi testate con il workload di Eurora disponibile, in diverse condizioni operative, e successivamente analizzate. Infine, si è contribuito in modo significativo allo sviluppo di diverse parti core del simulatore AccaSim.
Quest'elaborato mostra che l'impatto di diverse euristiche di allocazione sul throughput di un sistema HPC eterogeneo non è trascurabile, con variazioni in grado di raggiungere picchi di un ordine di grandezza. Tali differenze in termini di throughput sono inoltre molto più pronunciate se si considerano brevi intervalli temporali, come ad esempio dell'ordine dei mesi, suggerendoci che il comportamento a lungo termine del sistema è dettato principalmente dal metodo di scheduling utilizzato. Abbiamo inoltre osservato che l'impiego di euristiche per la predizione della durata dei jobs è di grande beneficio al throughput su tutte le euristiche di allocazione, e specialmente su quelle che integrano in maniera più profonda tali elementi data-driven. Infine, l'analisi effettuata ha permesso di caratterizzare integralmente il sistema Eurora ed il relativo workload, permettendoci di comprendere al meglio gli effetti su di esso dei diversi metodi di dispatching, nonché di estendere le nostre considerazioni anche ad altre classi di sistemi.

La tesi è strutturata come segue: nel Capitolo 1 presenteremo una breve panoramica dei sistemi HPC, mentre nel Capitolo 2 introdurremo formalmente il problema del dispatching, insieme alle soluzioni più comuni per lo scheduling e l'allocazione. Nel Capitolo 3 descriveremo il sistema Eurora, e successivamente il simulatore AccaSim, sviluppato ed utilizzato nell'ambito della tesi, nel Capitolo 4. Presenteremo dunque le soluzioni per lo scheduling e l'allocazione sviluppate nel Capitolo 5, e nel Capitolo 6 discuteremo i risultati sperimentali ottenuti con esse. Infine, nel Capitolo 7 presenteremo le nostre conclusioni, nonché la direzione del lavoro futuro.

Introduction

In the context of *High-Performance Computing* systems, good *dispatching* methods are a fundamental component that can help achieve good *Quality of Service* levels. By dispatching, we intend the methods with which tasks (or *jobs*) submitted by users to the system are selected and allowed to start on it, both in terms of *timing* and *allocation* of resources. In this work, we will focus on the design of **allocation heuristics** for dispatching; these will be targeted at *heterogeneous* HPC systems, which may possess different kinds of computing units in different nodes of the system. Some of our heuristics will be **data-driven** as well, thus exploiting information in the *workload* in order to estimate certain parameters and improve their own effectiveness.
Our analysis will be focused on **Eurora**, an heterogeneous HPC system developed by CINECA, in Bologna, and on a workload captured from its log trace, containing real user-submitted jobs: such a small-scale, heterogeneous context is a very good testbed for the evaluation of dispatching methods. All of our work was possible thanks to **AccaSim**, an HPC system simulator that we have developed in the Department of Computer Science and Engineering (DISI) of the University of Bologna: AccaSim is a novel instrument for the analysis of HPC systems, which has currently very few competitors in terms of flexibility and speed.

In detail, our work deals with the evaluation of HPC dispatching methods in a simulated environment, while using data-driven heuristics for the prediction of the jobs' duration. This was done in order to assess their impact on system *throughput*, in terms of job queue size and waiting times, again in the context of heterogeneous HPC systems, which are harder to manage than homogeneous systems.
The main contribution of our work lies with the design and development of new allocation heuristics: these were used together with some already-available scheduling methods, which were adapted and tweaked as well. The developed heuristics were then tested against the available workload for Eu-

rora, in a variety of conditions, and subsequently analyzed. At last, in our work we have significantly contributed to the development of various core parts of the AccaSim simulator.
This work shows that the impact of allocation heuristics on the throughput of an heterogeneous HPC system is not negligible, with variations that reach up to an order of magnitude in size. The differences in throughput between the various heuristics are also much more pronounced when considering short time frames, such as months, suggesting us that the system's long term behavior is dominated by the scheduling method being used. We have also observed that the usage of job duration prediction heuristics greatly benefits the throughput across all allocation heuristics, and especially on those that integrate such data-driven elements more deeply. Finally, our analysis helped fully characterize the Eurora system and its workload, allowing us to better comprehend the effect of various dispatching methods on it, and to extend our considerations to other systems as well.

The thesis is structured as follows: in Chapter 1 we will present a brief overview of HPC systems, and in Chapter 2 we will extensively discuss the dispatching problem, together with the most common solutions for scheduling and allocation. In Chapter 3 we will introduce the Eurora system, followed by the AccaSim simulator, which was developed and used in our work, in Chapter 4. We will then present all of the scheduling and allocation solutions that were developed in Chapter 5, and in Chapter 6 we will discuss the experimental results obtained with such heuristics. At last, in Chapter 7 we will present our conclusions, and point the direction of future work.

Contents

List of Figures

List of Tables

Chapter 1

An Overview of HPC Systems

In this chapter we will introduce the main notions behind HPC systems, and will provide basic knowledge which will be essential for the rest of the thesis. We will also look at the state of the art in HPC systems, and at the main architectural schemes used in this field.

The chapter is organized as follows: in Section 1.1 we will introduce the notion of HPC system. In Section 1.2 we will then look at the taxonomy of HPC systems, and at the architectural solutions that have emerged during the years. In Section 1.3, instead, we will present the state of the art in this field, while in Section 1.4 we will describe a generic architecture for systems of this kind.

1.1 HPC Systems and their Purposes

High-Performance Computing (HPC) defines a class of systems and problems, sharing the need for powerful computational resources and high flexibility [1]. While there is no formal definition, an HPC system is generally characterized by a custom-designed architecture, which offers computational resources to its users that are not commonly attainable. This necessity arises in turn from the need to perform very complex, data-intensive and resource-hungry tasks (or *jobs*) in a reasonably small time, which is very common

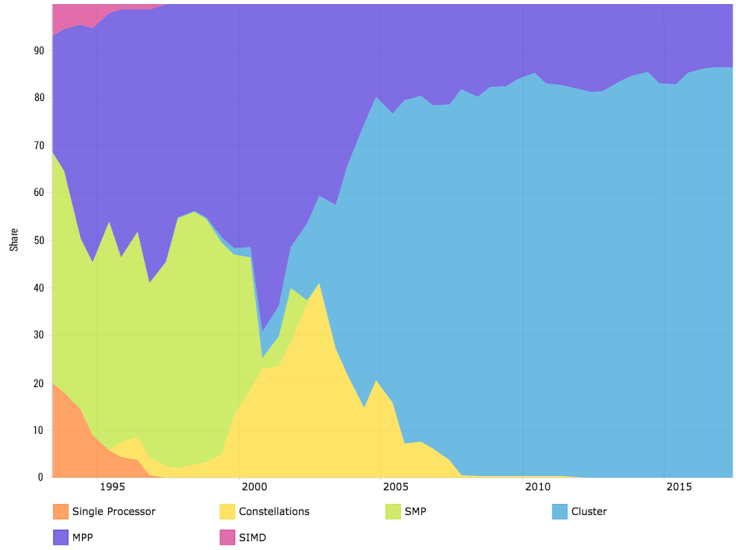


Figure 1.1: A plot of the market share for different HPC architectures over time. Image taken from [2].

today in research and industrial contexts.

The internal structure of an HPC system is usually hidden from the user, who interacts with it through specific interfaces. Such structure, however, is usually very complex and includes intricate networking, cooling and power infrastructures as the scale of the system grows, making the design and optimization of HPC systems an open research field.

HPC systems are used in a wide array of applications. Some of these, for example, are related to fields like *big data*, *complex systems*, *NP-Hard problem solving* and *scientific computing* in general: the number of fields in which there is a need for HPC systems and techniques is becoming higher and higher as technology advances.

1.2 Taxonomy of HPC Systems

Popular Architectures HPC systems come in different forms and shapes, and have greatly evolved in the last 20 years according to technological advancements [1], as it can be seen in Figure 1.1. In the past, HPC systems

mostly came in the form of *mainframes*, which were very powerful machines composed by just one node. These machines required specialized hardware and were highly expensive.
Modern HPC systems, and the dramatic improvement in terms of computing power that came with them, arrived however with the advent of *Cluster* architectures: with this term we refer to systems composed of a large number of inexpensive commodity machines. These machines can offer great combined computing power, and are also easy to replace in case of failure. In general, the resources in such a system are shared between many users as well. Being heavily distributed, Cluster systems are however burdened by many issues regarding connectivity, fault tolerance and power management. A variant of the Cluster architecture is the *Grid* one [3], in which machines in the system are geographically distributed, with network latency and node heterogeneity thus becoming critical aspects.
While our work is general and not bound to a specific system type, we will from now on focus on Cluster systems for our analysis.

Homogeneous and Heterogeneous Systems We may further divide HPC systems in two more categories, namely *Homogeneous* and *Heterogeneous* systems. As the name implies, an Homogenous system is one where there is only one type of main processing unit and one instruction set. This implies that all nodes are identical and made of the same components. In an Heterogenous system, instead, each node may possess multiple types of processing units and accelerators; for example, nodes may include GPU, FPGA or MIC units. Besides, nodes may be made of different components entirely. While heterogeneous HPC systems are generally more flexible than homogenous ones, they are also harder to manage, as the fragmentation of available resources may become a serious concern.

Online and Batch Systems HPC systems and computer systems in general can operate in *online* and *batch* modes: a system operating in batch mode will execute a pre-defined, static set of jobs at specific times. No new

jobs can be added without manual intervention, and finding the best order and resource assignment for the tasks to be executed is something that needs to be done only once. The jobs will complete in a finite amount of time, and the system will return to an idle state after that.
Online systems, on the other hand, are highly interactive and allow new jobs to freely enter at any time, without particular timing boundaries, and as such *they are always running*. This second class of systems, while much more powerful than batch systems, is also harder to manage as well, because of resource management concerns. Practically all HPC systems fall into the online category, and as such, we will treat techniques and algorithms designed for these systems.

1.3 State of the Art

At the time of writing (June 2017) the most powerful HPC system in the world is the **Sunway-TaihuLight**, located at the National Supercomputing Center in Wuxi, China [4]. This system can reach a peak performance of 125PFlops, and is made of more than 40000 computing nodes, grouped hierarchically at multiple levels. Each node contains an SW26010 unit, which is an integrated, heterogeneous, many-core processor, and 32GB of RAM. Power consumption under full load is measured at 15.371 MW.

Immediately after TaihuLight system we can find its predecessor, the **Tianhe-2** system, scoring 33.9PFlops of peak performance, and also located at China's National Supercomputing Center. It is composed of 16000 computing nodes, each having an Intel IvyBridge Xeon CPU, an Intel Xeon Phi co-processor, and 88GB of RAM. Other notable HPC systems are the **Piz Daint** and the **Titan**, with peak performances of 19.6PFlops and 17.6PFlops respectively [2].

As we can see, the most powerful HPC systems have broken the 100PFlops barrier, and are growing towards the *exa-scale* goal, which implies performance in the order of ExaFlops. This goal cannot be however reached by

just increasing the size of current HPC systems, as power consumption is a serious concern. To reach exa-scale performance, in fact, an increase of at least one order of magnitude in power consumption is required, compared to current HPC systems [5]; this means that the development of new techniques and architectures aimed at improving the energy efficiency and sustainability of HPC systems is now necessary more than ever.

1.4 Typical Structure of an HPC System

HPC systems may have wildly different software and physical architectures, depending on their nature and purpose: here, we will present the architecture of a generic cluster-based online HPC system.

1.4.1 Physical Architecture

In Figure 1.2 the physical architecture of the Sunway-TaihuLight system, introduced in Section 1.3, is shown. Being generic enough, this architecture will be used as a template to characterize how an HPC system usually works and arranges its physical resources.

Frontend Section The *frontend* of the system is made of a series of servers through which users can interact with it. Like in every large-scale system, such servers may be divided in different groups depending on their purpose, like *web access*, *system control*, *storage access*, and so on.

Backend Section We can find a *backend* section in the system as well. Such backend section is usually tasked with control and management of the system as a whole, and will include nodes for resource and state management, besides job dispatching.

Computing Nodes The *computing* section includes most of the physical resources in the system, and is made of the machines, or *nodes*, that actually

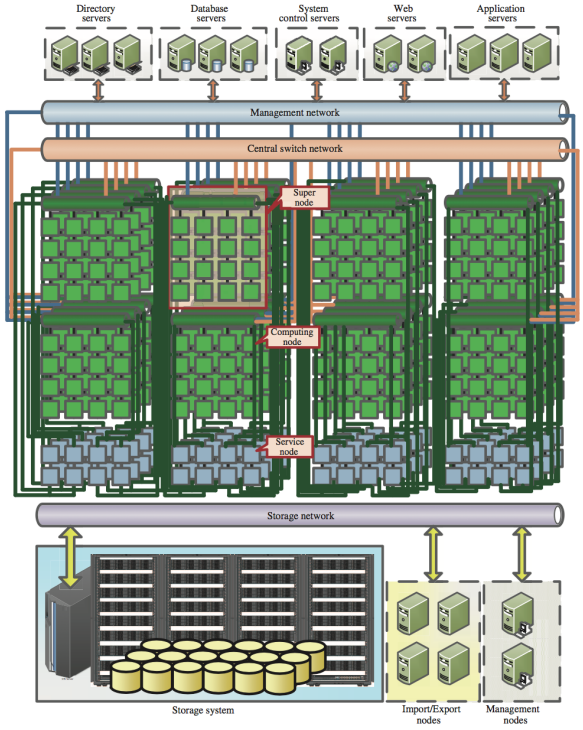


Figure 1.2: The physical architecture of the Sunway-TaihuLight HPC system. Image taken from [4].

execute jobs, and have the most computational power available. Computing nodes are usually organized hierarchically in sub-groups depending on the size of the system, physically and logically. This can lead to more efficient networking architectures, and can make the management of the system easier.

Networking Infrastructure *Networking* is a critical part in any HPC system. Different internal networks are usually employed for different purposes, such as for management or storage access. These networks are also arranged with a hierarchical structure reflecting that of computing nodes, thus with multiple levels of network switching.

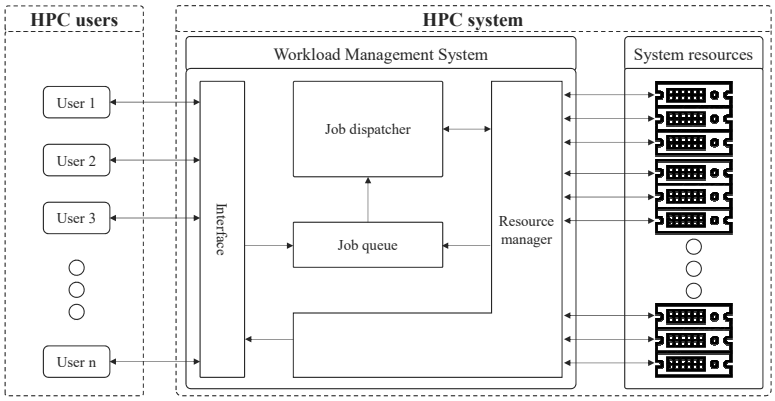


Figure 1.3: A simplified scheme for the software architecture of a generic HPC system. Image taken from [6].

1.4.2 Software Architecture

In Figure 1.3 we can see the typical basic software architecture used in modern HPC systems. The components we will now present are part of the *Workload Management System*, which is the component regulating the use of physical resources in the system, and the one we are most interested in.

User Interface Users interact with the system through a software *interface*, which acts as a frontend. This interface allows jobs to be remotely submitted by users, and regulates their arrival and execution; besides, it will also allow users to track their status, and manage them. It could be either a *GUI*-based or command-line interface. An HPC system usually also has one or more *queues* for jobs submitted by users that are waiting to be executed, with different priorities and features.

Dispatcher Actual management of jobs is handled by the *dispatcher* component, which periodically selects one or more jobs from the queues, and allows them to start on the system, either in a reactive or proactive way.
The dispatcher includes the *scheduler* and *allocator* components, which respectively define the *when* and *where* of each job's execution. In detail, the scheduler decides when a job should be executed, thus imposing the ordering

of jobs in the queue. The allocator, instead, decides which resources and computing nodes in the system a job should use. In general, the dispatcher component is the one responsible in the system with ensuring a good *Quality of Service* (QoS) level, by minimizing the waiting times for the submitted jobs. Besides deciding which, when and how jobs should run, the dispatcher is also tasked with triggering the start of these jobs by interacting with the *resource manager* component.

Resource Manager The dispatcher is able to start jobs and allocate them on specific nodes thanks to an abstraction layer called the *resource manager*: this entity keeps track of the state of resources and nodes in the system, besides that of running jobs. The resource manager usually acts as a server, with a monitor-like client running in each node in the system. Thus, when the dispatcher wants to execute a job, it will interact with the resource manager in order to perform scheduling and allocation, and then again to physically allocate the resources needed for the new job.

Chapter 2

Dispatching in HPC Systems

In this Chapter we will analyze the *dispatching* problem in HPC system, and see how it can be formalized. We will also discuss which are the most commonly used techniques and algorithms, and we will present some of the commercial solutions available on the market.

The Chapter is organized as follows: in Section 2.1 we will lay the formal foundations for the dispatching problem. In Sections 2.2 and 2.3, instead, we will discuss the *scheduling* and *allocation* sub-problems, respectively. Finally, in Section 2.4 we will look at the main commercial solutions for dispatching in HPC systems that are available in the market.

2.1 The Dispatching Problem

2.1.1 Formal Definition of Job

A *job* is, generically, a user-submitted task that is to be executed on an HPC system. Jobs in most HPC systems belong to the *parallel* class, and are composed of many independent running units that can communicate with each other, for example through *message-passing interfaces* [7].
A job J, made of an executable file together with its arguments and input data, is identified by an unique ID J_{ID}, and has various attributes associated

to it. Some of them are related to **time**, and the most important are:

- J_q: the *queue* in the system to which the job was submitted;
- J_{t_q}: the time at which the job was submitted to the system;
- J_{t_s}: the time at which the job started its execution;
- J_{t_e}: the time at which the job ended its execution;
- J_{d_e}: the *expected duration* for the job, which is an estimation and not representative of the real duration;
- J_{d_r}: the *real duration* for the job, which is computed after its termination as $J_{t_e} - J_{t_s}$;
- J_{d_w}: the *wall time*, which is the maximum time for which the job is allowed to run, and usually determined by the system itself;

A job may have other descriptive attributes, such as the name of the user that submitted it. There also are some attributes specifically associated to the **resources** requested by the job and supplied to it: a resource is the most elementary hardware unit of a certain kind available in a computing node, like a CPU core, or a certain quantity of RAM. These attributes are:

- J_n: the number of *job units* requested by the job. These can be considered as the independent instances, each of them running on a specific node, that make up the job;
- J_r: a data structure representing which types of resources are needed by each job unit, and in which amount. A *job unit request* is *homogeneous* if all J_n units request the same type and amount of resources, and *heterogeneous* otherwise;
- J_a: the resource *assignation* given by the system for the job, when it is scheduled to start. It can be interpreted as a vector of J_n records, with each of them containing the list of resources in a specific node assigned for a particular job unit.

According to how they behave in regards to the resources supplied by the system, we can additionally define various classes of jobs [7]:

- **Rigid**: these jobs need the exact amount of resources that were requested in order to run, and cannot adapt to any kind of change in their amount, either at run-time or at scheduling time;
- **Evolving**: unlike rigid jobs, they may request on their own initiative new resources to the system at run-time. If such new resources are not supplied by the system, the job won't be able proceed;
- **Moldable**: these jobs can adapt to an amount of resources supplied by the system that is higher or lower than the requested one; after the jobs starts, however, such allocation of resources is never allowed to change again;
- **Malleable**: they are a generalization of moldable jobs, and admit changes in the allocation of resources in respect to the requested ones both at run-time and at scheduling time.

Practical Assumptions

Job unit requests in PBS-based systems like Eurora, presented in Section 3.1, are all bound to be *rigid* and *homogeneous*, using J_n resource-wise identical job units. Since this is a reasonable assumption, and Eurora is our system of interest, we will also adopt this constraint, which has nothing to do with the heterogeneity of the underlying HPC system. Under this assumption, the J_r structure will be a list, with each element $J_{r,k}$ in it defining the amount of resources for type k needed by each job unit.
Also, the system's behavior regarding the wall time J_{d_w} may differ according to its nature: again, as in Eurora, we will suppose that any job exceeding in duration its wall time value will be terminated by the system.

Definition of Workload

Workloads are a very important part in the analysis of HPC systems. A workload is simply a set of jobs relative to a certain time frame, that need to be dispatched according to their submission time values and resource requests. A workload may be synthetic, and thus generated statistically by software, or extracted from log traces belonging to a real HPC system. In both cases, the jobs' real durations are included, allowing the workload to be used in simulated environments. Also, being inherently static, a workload allows for repeatable experiments and for reliable comparisons between multiple dispatching techniques. A workload is usually stored in a text-like file, with a certain format and specific attributes, with each entry corresponding to a single job.

2.1.2 Formal Definition of Dispatcher

The *dispatcher* is a software component in online HPC systems, which selects pending jobs from the queue, and allows them to start their execution. For simplicity, we will consider a system with only one queue available, but our considerations will be valid for systems with multiple queues as well. A dispatcher should be very fast and not computationally intensive, as it is the component responsible for guaranteeing a good QoS level in the system, and it can negatively impact its performance. The dispatcher is part of the Workload Management System, and it may behave in two different ways:

- **Reactive**: the dispatcher is invoked only when significant events occur in the system and trigger a status change, such as when new jobs arrive on the queue, or some others terminate;
- **Proactive**: the dispatcher is invoked independently from events in the system, for example in a *slotted* manner, thus dispatching jobs in the queue at regular time intervals.

The basic behavior of a dispatcher is shown in Equation 2.1. The dispatcher, when invoked, will allow a subset D of jobs waiting for execution in the queue Q to start. For each job J^i of D, the dispatcher will have assigned to it a starting time $J^i_{t_s}$ and a resource assignation J^i_a. This assignment is called a *dispatching decision*, and jobs in D that were scheduled to start at the current time step are prepared and removed from the queue.

In a dispatcher, the *scheduler* and *allocator* components can be identified: the scheduler is tasked to determine the starting time $J^i_{t_s}$ of each job in D and is identified by the s function, which may depend on both the characteristics of job J^i, and the status of the queue Q. The allocator, instead, defines a suitable resource assignment J^i_a for each scheduled job, and is identified by the a function, again depending on both the job J^i and the queue Q. Both the scheduler and the allocator interact with the *resource manager* component introduced in Section 1.4, in order to obtain information regarding the system's status.

$$\begin{aligned} D = (J^1, J^2, ..., J^n) \subseteq Q \\ J^i_{t_s} = s(J^i, Q),\ J^i_a = a(J^i, Q) \quad \forall J^i \in D \end{aligned} \tag{2.1}$$

The three phases of dispatching, namely job *selection*, *scheduling* and *allocation* are not to be intended in a sequential order. The subset D of jobs that are successfully dispatched is actually determined *after* the scheduling and allocation phases. In general, a *feedback loop* is present between the three parts of dispatching: the scheduler may select a particular subset of jobs to dispatch depending on its job selection heuristics, with certain starting times, and pass them to the allocator. However, the allocator may not be able to find suitable assignations for certain jobs, forcing the scheduler to select another strategy, or a different subset of jobs altogether.
The reasoning we presented is synthesized in Figure 2.1. In this figure, the solid lines represent the flow of execution, while dashed lines represent information used by entities in the dispatcher.

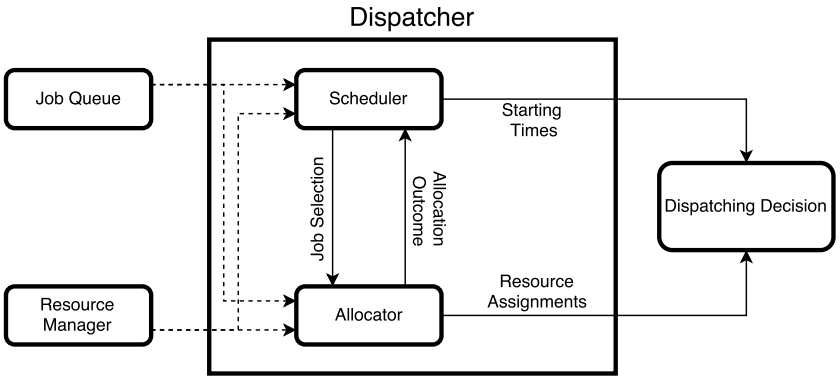


Figure 2.1: A representation of the workflow for a simple dispatching system.

2.1.3 Metrics for the Evaluation of Schedules

In this section we will present some of the many metrics generally used to evaluate the effectiveness of a dispatcher. These metrics consider different factors, and cover various aspects of the system: by using combining them, we can obtain a detailed view of the dispatcher's behavior, and make reasonable comparisons. As anticipated, a dispatcher is usually tested with a *workload* in order to evaluate its performance. This kind of methodology is preferable, as it allows for reliable and repeatable experiments.

All of the metrics we will present are mainly *comparative*: if a dispatcher achieved bad performance on a certain workload, it wouldn't necessarily mean that the dispatcher itself is not effective, but most likely that the workload is inherently difficult and hard to manage. At the same time, by using the same workload we can effectively compare various dispatching methods.
Finally, as most of the following metrics are computed on a *per-job* or *per-step* basis, it is usually necessary to look at their distribution for a given workload in order to obtain meaningful data on the system's behavior.

Makespan

The *makespan* is a temporal metric, and is very common for the evaluation of scheduling systems. It is formulated as follows:

$$mks = \max_{0 \leq i \leq n-1}(J^i_{t_e}) - \min_{0 \leq i \leq n-1}(J^i_{t_s}) \tag{2.2}$$

As represented in Equation 2.2, the makespan expresses the time interval between the *earliest* starting job, and the *latest* ending job. It is, in other words, an estimation of how effectively a dispatcher can *pack* jobs by assigning them to resources in the system: lower makespans signify better results, as it means the dispatcher is able to efficiently use the resources available in the system, which will be busy for a shorter time.
The makespan metric, however, is meaningful only for batch systems, that are bound to run for finite amounts of time; for online systems, instead, which are by definition made to be ever-running and prone to continuously-changing work conditions, the makespan is less relevant and not important.

Waiting Time

The *waiting time* is a temporal, *per-job* metric, and as the name implies it can be expressed through the following formula:

$$wait_J = J_{t_s} - J_{t_q} \tag{2.3}$$

As it can be seen in Equation 2.3, the waiting time for a job corresponds to the time interval between its arrival in the queue, expressed by J_{t_q}, and its starting time, represented by J_{t_s}. In general, lower waiting times can be associated with better results, as the system is able to promptly dispatch jobs without having them to wait too much time in the queue.

Sometimes, for systems with multiple job queues each with a different priority and *expected waiting time*, it may be useful to consider a normalized form of the waiting time, which is the *tardiness*: this metric corresponds to the job's wait divided by the expected waiting time for its queue, and is an approximation of the job's relative delay compared to how much it would have been expected to wait before being started.

Slowdown

The *slowdown* metric can be seen as a refined form of the waiting time [8]. The waiting time, in fact, does not consider one important fact: jobs with long durations are less susceptible to high waiting times, as they will have lower influence on the total *turnaround* time, represented by $wait_J + J_{d_r}$, which is the sum between the waiting and the real execution times. On shorter jobs, instead, the turnaround time could easily become greater than the execution time by orders of magnitude. The slowdown metric captures this, and can be interpreted as a sort of *perceived* waiting time. It is formulated as follows:

$$sld_J = \frac{wait_J + J_{d_r}}{J_{d_r}} \tag{2.4}$$

As expressed in Equation 2.4, the slowdown is computed as the turnaround time normalized by the real execution time of the job. A system achieving comparatively lower slowdown times is more performant: however, since the slowdown has a big impact on short jobs mainly, it is usually good practice to also consider the standard waiting time, which is equally representative for short and long jobs.

Throughput and Queue Size

The *throughput* and the *queue size* metrics are highly descriptive of a dispatcher's performance. These are not per-job metrics, but rather *per-step* ones: this means they are computed every time the dispatcher is invoked to schedule new jobs. The queue size and throughput metrics can be expressed through the following equation:

$$\begin{aligned} tp_t &= |D| \\ qs_t &= |Q| \end{aligned} \tag{2.5}$$

Equation 2.5 can be interpreted in a straightforward way. The throughput tp corresponds to the size of the set of jobs D that are successfully dispatched

and started at a certain time t, as seen in Equation 2.1. The queue size qs, instead, corresponds to the size of the queue Q after dispatching has been performed. As said earlier, both metrics are computed for every time step t in which the dispatcher is invoked. Also, the two metrics behave similarly: an higher throughput corresponds to a lower queue size, both of which mean the dispatcher is able to schedule jobs effectively. Considering just one of the two is usually enough to evaluate the performance of a system.

Resource Utilization

The most generic metric for the evaluation of a dispatcher in an HPC system is the *resource utilization*: it consists in the amount of resources in the system that are actively used by jobs at every time step. This is a per-step metric as well, but in this case we are considering all time steps, and not only those in which dispatching is involved. This is because jobs terminate their execution and free resources in the system independently from the dispatcher. A simple way to compute resource utilization is the following:

$$load_t = \frac{R_{used}}{R_{total}} \tag{2.6}$$

The formula depicted in Equation 2.6 defines the *load ratio* at time t, which is the ratio between the amount of used resources R_{used} in the system, and the total amount R_{total} of those available by default in it. This is a very good way to express the resource utilization, since the load ratio is represented by a number bounded in the range $[0, 1]$, thus independent from the scale of the system.
It is common to consider a specific subset of resource types in the system, in order to obtain more comprehensible and meaningful results: because of this, resource utilization is often computed in regards to CPU resources alone, since they are the most common ones and usually needed by all jobs.

In this case, looking at the distribution of resource utilization values may not be informative: using a visualization in function of time is much better,

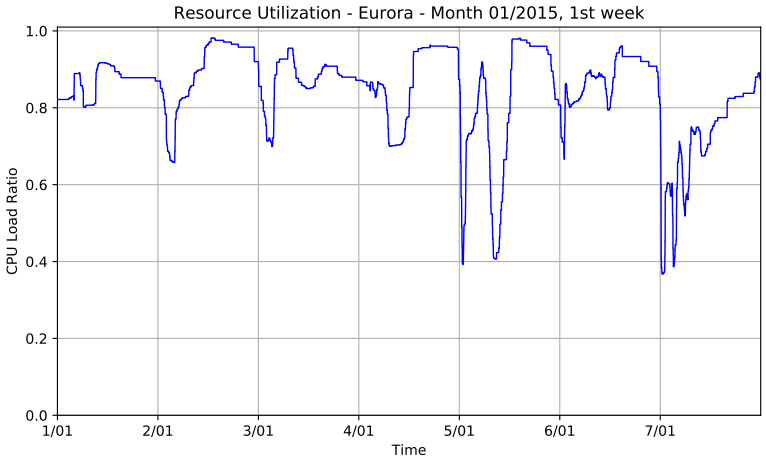


Figure 2.2: A plot depicting CPU resource utilization for the Eurora system, in the first week of January 2015.

and allows us to better compare different dispatching methods, as seen in Figure 2.2. The explanation is that the resource utilization metric is not meaningful at all times: during its normal operation, an HPC system will cross many working phases, some of which are not relevant. We are not, in fact, interested in periods when the system is in an *idle* or low-utilization state. We are only interested in observing the behavior of the system when it is almost fully loaded and there is a constant stream of jobs being submitted, as only in this scenario the qualities of a particular dispatcher can arise.

In general, low average resource utilization values and long queues indicate *fragmentation* in the system: with this term we mean the condition in which some nodes have few resources left, that cannot be used by any job and are thus wasted, and is mostly caused by badly designed allocation heuristics. Conversely, a dispatcher able to keep the system's resources fully loaded at most times is usually very good.

Resource Allocation Efficiency

Resource allocation efficiency is a metric that can be used to evaluate the performance of a dispatching system, in regards to its decisions [9]. It is a

per-job metric, and is formulated as follows:

$$eff_J = \frac{J_n * \sum_{k \in res} J_{r,k}}{\sum_{i \in J_a} R_{avl}(i)} \tag{2.7}$$

In Equation 2.7, the upper member in the fraction represents the total amount of resources needed by a job, with J_n being its job units, and $J_{r,k}$ the amount of resources of type k needed by each of them. J_a is instead the list of distinct node assignations for job J, and $R_{avl}(i)$ is the amount of available resources in such nodes before dispatching.
The resource allocation efficiency allows us to estimate how efficiently a dispatcher can allocate jobs in the system: high values indicate that a job uses few nodes, and that these nodes are used to their fullest, leaving no free resources, and thus also implying low fragmentation. An highly efficient dispatching system will lead to higher throughput, and to lower power consumption as well.

This metric can be formulated on a per-step basis as well: in this case, at each time step all the running jobs and the nodes on which they are allocated are considered. This variant of the resource allocation efficiency metric is mostly related to the system utilization over time, and can be seen as a refined form of the load ratio.

2.2 The Scheduling Sub-Problem

2.2.1 Definition and Hypotheses

In HPC systems, the *scheduling* problem consists in assigning starting times J_{t_s} to a series of jobs, by using heuristics in order to maximize the resource usage of the system, and minimize the waiting times. At this stage we are not assigning resources to jobs, but only determining, if necessary, wether they *fit* the current available resources in the system or not: which of those will be assigned to them is up to the allocator to decide.

In general, there are two possible approaches to scheduling in HPC systems [10]. These are:

- **Queueing**: the scheduler will use a certain ordering criteria for jobs in the queue, and will try to schedule as many of them as possible every time the dispatcher is invoked, in a sequential order: hence, jobs are always given an immediate starting time, corresponding to when the scheduler made its decision. When the scheduler reaches a job that cannot be dispatched, because there are not enough available resources in the system, it will terminate, as not to violate the properties of the queue. This is the most common approach; it is also very simple, and not computationally complex.

- **Planning**: every time the dispatcher is invoked, the scheduler will try to compute a *schedule plan* for all jobs in the queue, or a subset of them; that is, a specific starting time is assigned to each job, without violating the system's resource constraints. This is done in order to find the globally best possible placement for jobs, in terms of waiting times or other metrics, which simply cannot be done by a queueing-oriented scheduler. Unfortunately, planning-oriented scheduling algorithms are inherently complex, as the problem itself of assigning starting times to tasks with certain constraints belongs to the NP-Hard class [11]; besides, in order to compute such a schedule and obtain good results, it is necessary to have reliable estimations J_{d_e} of the jobs' length: this is often not the case, and all we have is the wall time J_{d_w}, which severely over-estimates the job length and leads to resource waste.

Out of the two presented approaches, the queueing one is the most commonly used in HPC systems, and in most commercial solutions for dispatching. The planning approach, instead, while being potentially better is plagued by its computationally intensive nature, and thus more rarely used.

2.2.2 Common Scheduling Algorithms

In this section we will discuss the most common algorithms for scheduling in HPC systems. Please note that these methods all belong to the queueing-oriented kind, except the last one.

First-Come First-Served

The first and most simple scheduling algorithm we will present is *First Come First Served* (FCFS). As the name implies, this method does not impose any kind of ordering on the job queue, and the scheduler will try to dispatch jobs in the order in which they arrived.
This algorithm has some qualities: first of all, as we mentioned earlier, it is very simple and computationally inexpensive. Also, since jobs are not artificially sorted, it ensures *fairness* in the scheduling process, and it is not possible for some jobs to suffer *starvation*. Finally, it does not need any kind of information about the system or the jobs in order to perform its dispatching decisions. However, since no kind of optimization is performed, FCFS may pick jobs in a highly sub-optimal order, thus leading to bad performance, low resource usage in the system, and long waiting times.

Longest Job First and Shortest Job First

The *Shortest Job First* (SJF) and *Longest Job First* (LJF) algorithms are the natural evolution of FCFS. In simple words, these two methods will sort jobs in the queue by using their estimated duration in ascending (SJF) or descending order (LJF). These are slightly more computationally expensive than the FCFS algorithm, however the computational cost can be reduced by maintaining the job queue in a sorted state between dispatching calls.
The choice between SJF and LJF is not obvious, and mainly depends on the kind of workload the system is subject to. However, among the two, SJF is usually the most robust choice, and leads to good results in terms of throughput and waiting times, despite being very simple.

Unfortunately, these two algorithms still have some issues to be considered: first of all, since sorting is explicitly performed on the job queue, we must consider the possibility of *starvation* for some jobs. For example, in a system employing SJF where the frequency of short jobs is very high, a long job may end up waiting indefinitely in the queue. Secondly, both methods need an estimation J_{d_e} of the jobs' length in order to perform sorting: if there is not one available, the wall time J_{d_w} must be used, leading to worse results.

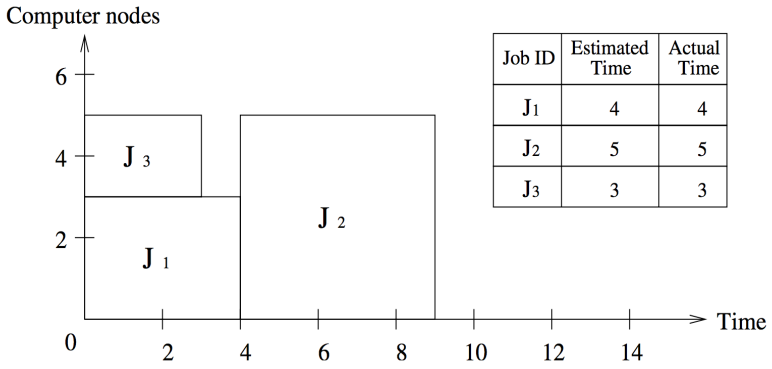
Priority Rule-Based

Priority Rule-Based (PRB) scheduling is a generalization of the FCFS, SJF and LJF algorithms seen before [12]. It still uses a simple queueing-oriented approach, but in this case the sorting criteria for the jobs is a generic *priority rule* that can be changed and tuned according to the users' needs and the system's type. For example, in a system with multiple job queues, each with a specific priority and *expected waiting time*, a priority rule could be based on a job's *tardiness*, introduced in Section 2.1.3 [7].
As with the methods we have seen before, caution must be taken while designing new priority rules: the risk of favoring certain classes of jobs while compromising others, with certain workload distributions, is always present.

Backfill

Backfill is a very commonly used queueing-oriented scheduling technique, and is an industry standard in HPC systems. Generally speaking, backfill is not a technique made to replace the heuristics we have presented so far, but rather it can be placed *on top* of them, as it does not make any assumptions on job ordering.

In a standard scheduling algorithm, like the ones we have seen earlier, jobs are scheduled in a sequential manner depending on the queue's ordering. This is what we would call the *normal* mode of the scheduler. Whenever the allocation for a job fails, the scheduler terminates, and returns the set of jobs



Job ID	Estimated Time	Actual Time
J_1	4	4
J_2	5	5
J_3	3	3

Figure 2.3: The representation of a schedule produced by an Easy backfill algorithm. Image taken from [13].

that it managed to schedule. In a backfill scheduler, instead, the procedure does not terminate when a job cannot be allocated: instead, a *reservation* is made for it. Specifically, the algorithm will scan through the set of currently running jobs, and from their estimated duration J_{d_e} it will compute the earliest starting time in which enough resources will be available in order to start the blocked job. Then, a set of resources in the system is reserved for that specific starting time and for the job's expected duration.
As long as the reservation has not been fulfilled, the scheduler operates in the homonymous *backfill* mode: in this special mode, the scheduler will try to dispatch jobs in the queue other than the blocked one, as long as they do not interfere with the reservation; in particular, they must not use any of the resources in the reserved set, in the time frame of the blocked job's planned execution. Also, in backfill mode the scheduler is allowed to *skip* jobs in the queue that cannot be dispatched.

What we just described is the basic functioning of the backfill algorithm. There are, however, two different variants of this technique:

- **Easy Backfill**: only one reservation at most is maintained at any time. This reservation corresponds to the blocked job, which is located at the head of the queue;

- **Conservative Backfill**: multiple reservations are admitted, and created whenever a job cannot be scheduled. In this case, there is no real functional distinction between the normal and backfill modes.

It has been shown that there is no clear winner between the two Backfill modes, and their behavior mainly depends on the kind of workload they are subject to [13], even though Conservative is much more complex than Easy Backfill. In Figure 2.3 a representation of how Easy backfill works can be seen, with J^2 being the blocked job.

Finally, it must be said that since backfill relies on estimations for the jobs' length, its performance heavily depends on these: in fact, an *overestimation* of the jobs' length might lead to wasted resources in the backfill interval, while an *underestimation* could lead either to preemption of the jobs that didn't manage to finish before the end of the backfill interval, if the system supports it [14], or to a delay of the reservation itself.

Optimization and Planning-Oriented Algorithms

In this last section we will look at how the scheduling problem can be formulated in order to pursue an optimization and planning-oriented approach. In general, an algorithm of such kind will operate on a subset of the job queue of fixed maximum size, which will be sorted according to some priority rule. As mentioned earlier, for these jobs a *schedule plan* is computed, detailing the starting times for each one of them. As such, this is the only case in which the subset D of jobs that are to be dispatched is determined *before* the scheduling procedure.
In order to formalize the problem, we will now define its parameters:

- **Variables**: the starting times J_{t_s} for jobs in the subset D of the queue being considered;

- **Constraints**: the schedule plan must never violate the system's resource availability constraints;

- **Objective Functions**: a certain metric that is to be minimized in order to obtain an optimal solution, such as the *makespan* or the *average waiting time*.

This is a general characterization for the scheduling problem, which is valid for most applications in HPC systems. Having defined its parameters, a method for solving the problem has to be chosen: since it belongs to the NP-Hard class, exhaustive search must be excluded from the viable alternatives, as it would be unfeasible on an online system. Besides, a sub-optimal solution is often more than enough for this kind of task. As such, optimization techniques like *simulated annealing*, *genetic optimization*, *tabu search* or *constraint programming* are much preferable. In general, these techniques can be adapted to the timing constraints of the system by applying temporal limits to the search, with better solutions more likely to be found as the search time is increased.

As we have seen with the previous algorithms, being able to correctly estimate the jobs' length is critical: wrong estimations will result in the computed schedule plan to not be respected, thus worsening the performance of the algorithm. At the same time, an approach must be defined in regards to new jobs arriving in the queue. The algorithm might be designed to preserve an already-computed schedule plan, and *fit* the new jobs inside it at the next dispatching call. Conversely, it might be preferable to recompute the schedule plan from scratch every time, leading to potentially better results but also to an higher computational cost. An algorithm belonging to this class will be described with great detail in Section 5.2.4.

2.3 The Allocation Sub-Problem

2.3.1 Definition and Hypotheses

In this section we will treat the *allocation* sub-problem in dispatching. This consists in assigning a set of resources J_a, according to their requested job

units J_n and resource types J_r, to jobs in the subset D of the queue that were selected by the scheduler and given a starting time.
Allocation is a task of great importance in an HPC system. Good allocation policies allow to greatly improve parameters of the system like power consumption and average temperature, but also allow for lower resource fragmentation and, in turn, lower waiting times.
We can distinguish between two main approaches for allocation in HPC systems [15]:

- **First-Fit**: this approach is similar to queueing-oriented scheduling. In this case, jobs are allocated one by one separately, in the order specified by the scheduler. For each job, resources are picked from a list of nodes, which is sorted according to a certain criteria: the algorithm will pick resources from the list while traversing it, until the job's request has been satisfied. Usually, the algorithm will try to fit as many job units as possible in each selected node. If the algorithm reaches the end of the list without finding enough resources, the allocation is considered as failed, and the scheduler must then decide how to proceed;

- **Mapping**: conversely, this approach is equivalent to planning-oriented scheduling. Here, jobs in D are considered as a whole and collectively allocated, using complex algorithms that try to optimize specific metrics [15]. Like the scheduling problem, also the *mapping* problem, which consists in assigning resources to a given set of jobs in an optimal manner, belongs to the NP-Hard class. Besides, most mapping methods are made to be used on their own, without depending on a scheduler, as they can decide the jobs' starting times as well: in these cases, our proposed architecture for a dispatcher, seen in Section 2.1, is not applicable.

In most HPC systems, allocation algorithms belonging to the first-fit class are used. Mapping algorithms, while potentially better, are also much more complex, and usually need specialized software architectures in order to be

correctly integrated with scheduler algorithms. For these reasons, we will not further discuss mapping algorithms.

2.3.2 Common Allocation Algorithms

In this section we will present some allocation methods that fall into the first-fit category, and are very commonly used in HPC systems. Most of the algorithms in this section perform a *fail-first* search: this means that the ordering of nodes is such that a suitable allocation is more likely to be found on later nodes in the sorted list, rather than the first ones. This is due to the fact that such type of search, while theoretically more expensive than a success-first one, usually allows for better results.

Simple First-Fit Policy

The first algorithm for allocation we will present is the *simple first-fit* policy: similarly to the FCFS scheduling policy presented in Section 2.2.2, this method does not perform any kind of sorting, and nodes are scanned for available resources in their default order. This order may be numerical or lexicographical basing on each node's ID, but it may also be a static ordering made to improve certain performance parameters.

While simple, this algorithm has not inherently bad performance; it is in fact very common in HPC systems. It also has a few interesting properties: since nodes are always scanned in the same order for all jobs, the system's resources will statistically be filled in an incremental manner. This means that before moving to the next nodes in the list, the previous ones will usually have reached maximum load, leading in turn to low resource fragmentation.

Best-Fit Policy

The *best-fit* heuristic is an improvement over the basic first-fit allocation method. In this case, nodes in the system are actively sorted according to the amount of resources available in them, in ascending order. This means

that the first elements in the nodes' list will usually correspond to ones that have none or very little available resources.
The purpose of this is to decrease fragmentation in the system and thus perform *consolidation*. The first-fit policy is not enough for this, as jobs will terminate and release their resources in arbitrary order, which means that, when the system is fully loaded and running, the allocator's performance may become completely random. The best-fit policy addresses this, ensuring that at every allocation the best-fitting nodes are selected, keeping fragmentation low.

For complex, sorting-based allocation algorithms like best-fit performance may be a concern: systems may in fact scale horizontally indefinitely, and could be made of thousands of nodes. Performing sorting on the nodes' list at every allocation is thus not very efficient. To address this, there usually are two possible ways: a persistent, sorted list of nodes in the system may be kept, which will drastically decrease the computational cost of the algorithm; in alternative, smaller subsets of nodes in the system may be considered for allocation, by using for example tree-based selection techniques.

Priority Rule-Based

As we have seen with the scheduling problem, there also exists a *priority rule-based* generalization for allocation algorithms. In this case, the order of nodes in the system picked for allocation depends on a user-defined priority rule, which may take several factors into account. Again, the development of priority rules is not a trivial task, and without mindful design it can lead to badly performing allocators.

Such an allocation heuristic, for example, may be related to heterogeneous systems equipped with multiple accelerator types: a priority rule could *weight* different resource types, assigning greater weight to those that are scarce in the system and not available in every node, in order to not penalize jobs that actually need them. The nodes containing such resources would then

be statistically placed towards the end of the nodes' list, thus preserving the critical resources.

Cooling-aware and Power-aware Placement

Some allocators in the literature are aimed at minimizing the system's *temperature* and *cooling power* [16]. Cooling systems are in fact a very important component in HPC systems, and managing to keep the overall system temperature low will lead to better performance and to more efficient power consumption. These techniques go under the name of *cooling-aware placement*, and it is estimated that, in an optimal scenario, the use of such algorithms can reduce the costs for environmental management in an HPC system by up to 30% [17].

Minimizing the overall system's temperature increase after the allocation of a job implies the use of complex optimization and heuristic techniques, besides models for temperature prediction in specific parts of the system. Also, in order to keep track of the system's temperature in all nodes, an additional hardware-software infrastructure is necessary.
Having said this, such a type of allocation is usually obtained by placing jobs in nodes that are physically far from each other, in order to evenly distribute the temperature increase; in fact, placing all job units in nodes close to each other would cause a spike in the temperature for that area, which would require higher cooling power.

Cooling-aware placement can help reduce overall power consumption: however, that is often not enough, and specialized *power-aware placement* techniques are needed. As we have seen, the first-fit and best-fit algorithms are able to keep fragmentation in a system low: this is a good starting point for power consumption optimization, as having nodes either in a *fully loaded* or *idle* state is an ideal condition. This way, many nodes are able to enter special, low-power idle modes, that can drastically improve the overall energy efficiency.

Similarly to what we have seen with cooling-aware placement, there are complex power-aware placement algorithms, which try to minimize the overall power consumption increase after the allocation of a job [18]; again, complex models for power consumption prediction in regards to the system's hardware components must be used.

Topology-aware Placement

Often the physical placement of a job, independently from the resources available in its assigned nodes, may have a big impact on its performance. Parallel jobs, in fact, usually have intricate communication patterns: this means that the farther away job units are placed from each other, the more network hops are needed for communication. This implies higher network strain and latency times, which both lead to worse performance on the job's side, and higher power consumption. For this reason, some allocation methods try to place units of the same job in a physical area as small as possible, trying to improve the *locality* of a job.

In order to perform this kind of allocation, it is necessary to know the topology of the system. Many such algorithms are known in literature [19], and the allocation policy used in Slurm also exploits locality [20]: these methods map nodes in a tree, with each level signifying different grouping hierarchies for nodes. Leaf nodes sharing the same parent generally belong to the same basic grouping unit, which may be a rack or a cabinet. This kind of approach allows for efficient tree-based search techniques, and is most effective on large-scale systems.

2.4 Commercial Workload Management Systems

In this section we will present some of the most famous commercial HPC Workload Management Systems, and their peculiarities.

The first solution we are looking at is **Slurm** [20], which is a free, open source HPC dispatching system targeted for Linux/Unix. Slurm is maintained on GitHub, and has an active community behind it. Due to its highly modular and customizable nature, Slurm is used in roughly the 60% of HPC systems in the world, including the Tianhe-2 system introduced in Section 1.3. Besides job scheduling and resource management, Slurm has many other features, mainly for system monitoring and control. It is advertised as an highly scalable system and as easy to configure, thus useful in many contexts. Slurm uses a topology-aware allocation policy, trying to improve locality and resource utilization, and supports heterogeneous job unit requests as well.

The second WMS we present is **Portable Batch System** (PBS) [21], which is a commercial product made by *Altair*. This is the dispatching system also used in Eurora, and is an highly reliable product that has been on the market for 20 years in different iterations, even though originally it was free and open source. Many other products in the market are based on PBS, such as Torque, which we will describe later. Compared to Slurm, PBS has also power-aware capabilities, however it is far less modular and harder to customize. Unlike the former, PBS also does not support heterogeneous job unit requests, which means a PBS job is limited to having J_n homogeneous job units, each requiring the same amount and type of resources.

Torque [22] is a WMS based on the original open-source PBS project. It is produced by *Adaptive Computing*, but is free and open source, with the company being tasked with user support and development. Similarly to Slurm, Torque is highly modular, scalable and customizable, and can be adapted to a great variety of systems, with a focus on heterogeneous ones.
At last, we will talk about the **Maui** system [23], which is right now also under the custody of *Adaptive Computing*. Maui is mostly a predecessor to modern dispatching systems, and was mainly developed during the 90s. It provided customizable fairness, job priority and allocation policies, which are a standard in modern products. It is now discontinued, and its core was inherited by the **Moab** system, now still actively developed.

Chapter 3

The Eurora System

In this chapter we will introduce the Eurora system, which was chosen in order to evaluate the various dispatching methods that will be presented later. We will analyze a workload from Eurora as well, in order to better understand its usage patterns, and finally we will discuss some methods to estimate the jobs' durations on such workload.

The chapter is structured as follows: in Section 3.1 we will introduce the Eurora system, and its main features. In Section 3.2 we will then analyze the workload from Eurora that will also be used for testing later in the thesis. Finally, in Section 3.3 we will discuss a data-driven method for the estimation of the jobs' durations in the workload.

3.1 System Overview

Eurora is a prototype HPC system built in 2013 by CINECA in Bologna, Italy, in the scope of the *Partnership for Advanced Computing in Europe* (PRACE) [24], and is pictured in Figure 3.1. It is a small-scale HPC system with an hybrid architecture, designed for low power consumption: Eurora was in fact listed as #1 in the Green500 list of Top500, which ranks the most efficient HPC systems worldwide, in July 2013 [2]. It could achieve a 3.2 GFlops/W computing power, and had a peak power usage of 30.7KW.

Figure 3.1: A picture of the Eurora system. Image taken from [24].

The Eurora system is a heterogeneous cluster made of 64 nodes: each of these nodes has two Intel Xeon SandyBridge CPUs with 8 cores, 16GB of RAM, and 100TB of disk space. Additionally, each node has two accelerator units available: specifically, 32 nodes are equipped with two NVIDIA Tesla K20 GPUs, while the remaining 32 are equipped with two Intel Xeon Phi *Many-Integrated-Cores* (MIC) units. Some nodes differ slightly in terms of CPU and RAM: one half of the nodes, in fact, uses CPUs clocked at 2.0Ghz, while the other half uses CPUs with a clock of 3.1Ghz. At the same time, 6 nodes in the system, with the higher performance CPUs, mount 32GB of RAM storage instead of 16.

The system's network has the topology of a 3D torus, and networking tasks in each node are handled by an Altera Stratix-V FPGA unit and by an InfiniBand switch operating in *Quad Data Rate* mode. InfiniBand is a standard for computer networking with very high throughput and low latency, and is commonly used in HPC systems [25].

The nodes run a Linux CentOS 6.3 distribution, and the workload management system being used is *Portable Batch System* (PBS), which employs various heuristics for optimal throughput and resource management; since

Queue	Max. Nodes	Max. Cores/GPUs	Max. Time	Approx Wait
debug	2	32 / 4	00:30:00	Seconds
parallel	32	512 / 64	06:00:00	Minutes
longpar	16	256 / 32	24:00:00	Hours

Table 3.1: The constraints related to each of the job queues in Eurora.

PBS is being used, jobs in Eurora are limited to homogeneous job unit requests, as mentioned in Section 2.4. Additionally, Eurora uses three different queues for job dispatching [7], named *debug*, *parallel* and *longpar*, with different priorities and resource constraints. The *debug* queue is designed for quick, small jobs executed for debug purposes; the *parallel* queue is instead designed for ordinary jobs, while the *longpar* queue is made for long, low priority jobs that are to be scheduled during the night. The specifics for each queue can be seen in Table 3.1.

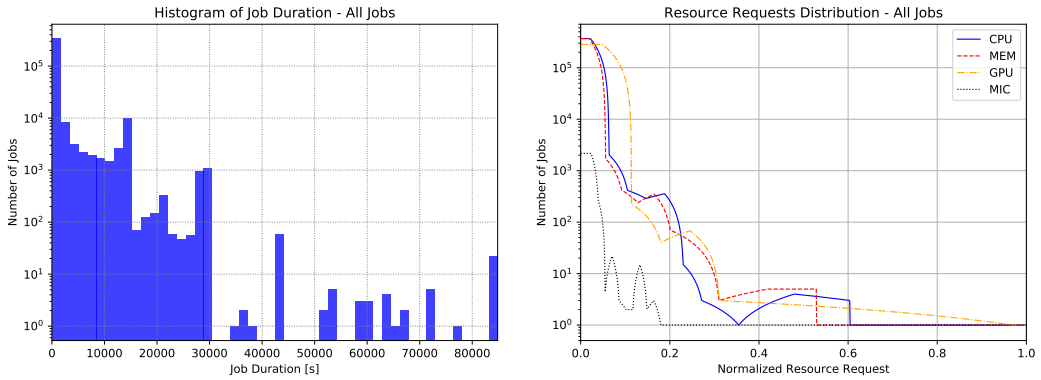
The heterogeneity seen in the available resources grants great flexibility to Eurora, and also makes it an interesting system to analyze, especially due to its limited scale. In such a context, the dispatcher component has primary importance, in order to make good use of the system: for these reasons, Eurora will be the object of our analysis in the scope of new dispatching heuristics development.

3.2 Analyzing the Eurora Workload

We will now consider a workload extracted from Eurora's log traces, and analyze it thoroughly from different points of view. The workload will be used for testing later in the process, with the AccaSim simulator that will be presented in Chapter 4.

3.2.1 Workload Overview

The workload is made of 372320 jobs and covers one year of data, from April 2014 to June 2015. As mentioned earlier, in Eurora there are GPU and



(a) Job duration distribution. (b) Resource requests distribution.

Figure 3.2: Job Duration and resource requests distributions for all jobs in the workload.

MIC accelerators available in some nodes, so it will be useful to consider different classes of jobs according to the resources they use. In particular, we will consider *Standard* jobs, which use only CPU and memory resources, *GPU-Based* jobs, which also use GPU resources, and finally *MIC-Based* jobs, which use MIC accelerators. It is not possible for a job to use both GPU and MIC accelerators, since a node may have only one of these available at a time, and PBS does not allow heterogeneous job unit requests.

In Table 3.2 we can see some statistics for the workload. It can be seen that GPU-Based jobs are the most frequent ones, making up over 75% of the entire workload. Standard jobs instead have a share of roughly 22%, while MIC-Based jobs have a very marginal role, amounting to just 0.7%.
It can also be seen that the average real job duration is low, amounting to 16 minutes. Predictably, such a low value is due to the GPU-Based jobs, which have an average duration of just 6 minutes, while MIC-Based jobs tend to be much longer, with a value of 56 minutes. Standard jobs, instead, are mostly in the middle ground, with a value of 47 minutes. We can see that the maximum duration value belongs to a GPU-Based job, with 23:33 hours, close to the 24 hours wall time limit in Eurora. This value is mostly an outlier in the distribution, and not really meaningful about the duration of GPU-Based jobs.

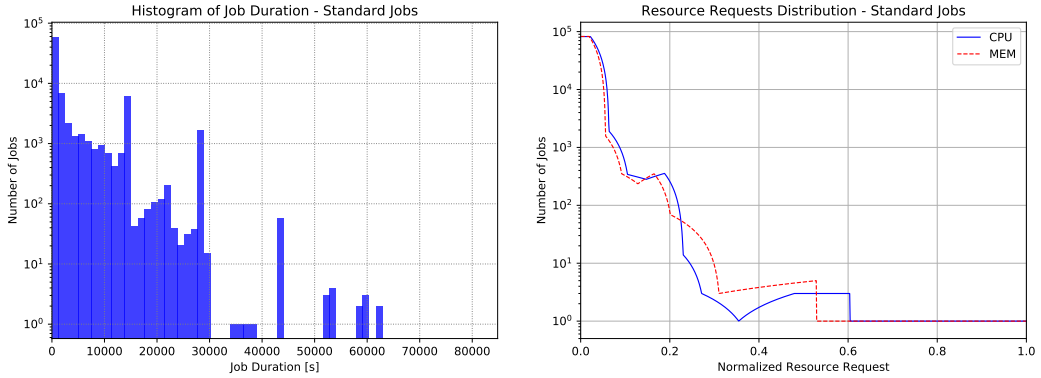
Job Types	Share	Number	Avg. Duration [hh:mm:ss]	Max. Duration [hh:mm:ss]
All	100%	372320	00:16:08	23:33:54
Standard	22.8%	85046	00:47:36	17:28:38
MIC-Based	0.7%	2500	00:56:28	08:12:26
GPU-Based	76.4%	284774	00:06:23	23:33:54

Table 3.2: Statistics for different kinds of jobs in the Eurora workload.

In Figures 3.2a and 3.2b we can see, respectively, the distributions for job duration and resource requests in the workload. The durations follow an heavy-tailed distribution, with most jobs being relatively short and few of them reaching close to the maximum limit. Because of this, we expect scheduling policies such as Shortest Job First to behave really well with this workload.
The jobs' resource requests distribution has an heavy-tailed behavior as well, and most jobs in the workload are not particularly resource-hungry. For this reason, Backfill may be a very effective scheduling policy as well. It should be noted that in this second plot, we are considering the *total* amount of resources requested by a job. The resource requests of each job are then normalized against the total amount available in the system for a given type, in order to obtain an homogeneous visualization across all resource types. For this same reason, we can see that the CPU resource request distribution is truncated closely to the 0.6 value: this is because, in Eurora, jobs are not allowed to use more than 512 cores in the system against the 1024 available, as explained in Section 3.1.

3.2.2 Detailed Analysis

We will now analyze with this same approach the single classes of jobs introduced earlier, and see if we can find any peculiarities in them.



(a) Job duration distribution. (b) Resource requests distribution.

Figure 3.3: Job Duration and resource requests distributions for Standard jobs in the workload.

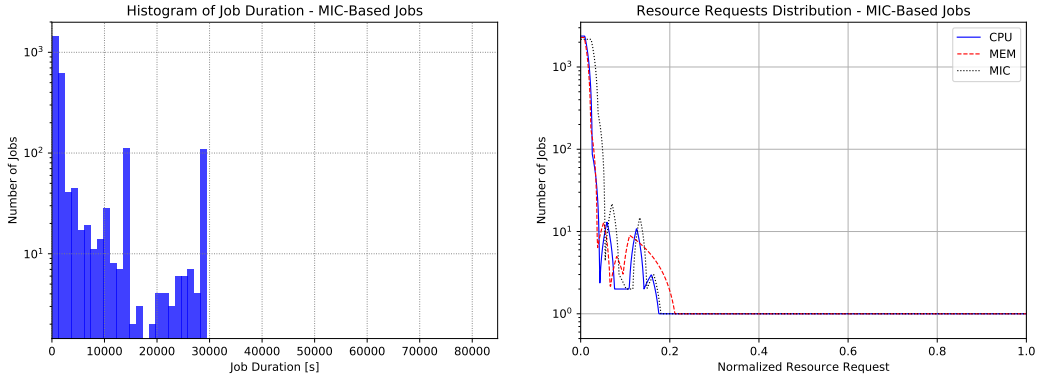
Standard Jobs

Standard jobs, as said earlier, only use CPU and memory resources, and make up roughly 22% of the Eurora workload. In Figures 3.3a and 3.3b we can see, again, their distributions for job duration and resource requests respectively. It can be seen that these distributions do not differ in a meaningful way compared to the ones presented earlier, for all jobs. This is to be expected, especially for the resource requests distribution, as Standard jobs are heavily based on CPU and memory resources, and thus have a strong influence on the global distribution, proving responsible for its heavy-tailed behavior.

Since these jobs can fit all nodes in the system and do not have special needs, we won't need to adopt particular policies in order to improve their QoS.

MIC-based Jobs

We will now consider MIC-Based jobs in Eurora, which use CPU, memory and MIC resources, and constitute only the 0.7% of the entire workload.
In Figures 3.4a and 3.4b, we can see the job duration and resource requests distributions for this class of jobs. While similar to the ones we have previously seen, the job duration distribution has a slightly fatter tail: this results



(a) Job duration distribution. (b) Resource requests distribution.

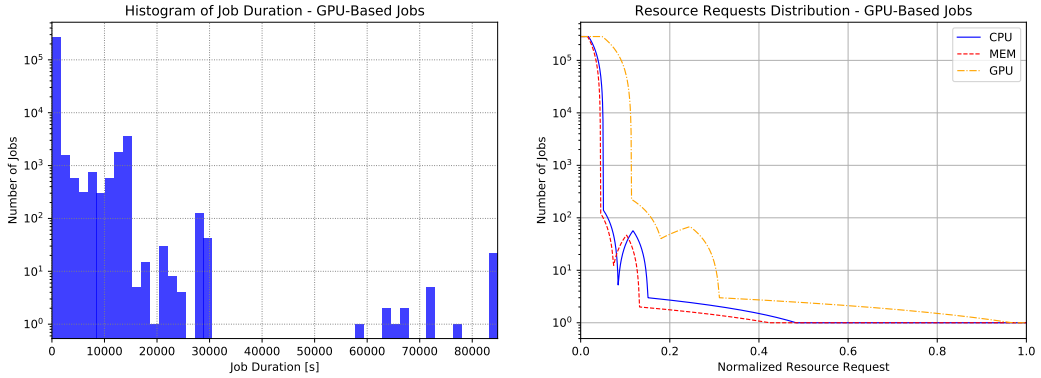
Figure 3.4: Job Duration and resource requests distributions for MIC-Based jobs in the workload.

in longer average job durations, as we have seen in Table 3.2. This is counter-intuitive, as accelerator-based jobs would be expected to last much less than Standard jobs. However, there are no jobs longer than roughly 8 hours.
The resource distribution is, instead, much more concentrated on the left side, and does not exhibit an heavy-tailed behavior. This means that MIC-Based jobs generally require very few resources and, interestingly, there are no jobs requiring more than 20% of the MIC units available in the system.

MIC-Based jobs may be difficult to manage: while not extremely long, they have an high average duration, and they rely on a resource type that is scarcely available in the system. Also, since the frequency of MIC-Based jobs is so low, it is very likely for high fragmentation to occur for MIC resources, as their nodes would be filled by Standard jobs most of the time. This could result in extremely high waiting times for MIC-Based jobs.

GPU-based Jobs

We will at last consider GPU-Based jobs, which use CPU, memory and GPU resources. This is a very important class of jobs, as it makes up more than 75% of the workload. The corresponding job duration and resource requests distribution can be seen respectively in Figures 3.5a and 3.5b. As for the



(a) Job duration distribution. (b) Resource requests distribution.

Figure 3.5: Job Duration and resource requests distributions for GPU-Based jobs in the workload.

job duration distribution, it can be seen that while there is an heavy-tailed behavior, with few jobs having very high durations, the distribution is highly concentrated on the left side, thus resulting in very low average job durations. The job resource distribution, similarly to the MIC-Based jobs one, is also highly leaning to the left side, meaning that most GPU-Based jobs are small in terms of resources. However, there is indeed an heavy-tailed behavior here, meaning that a small subset of jobs will need a large amount of resources, with one particular job needing all GPU units available in the system.

GPU-Based jobs are absolutely critical in Eurora. They make up the biggest part of the workload, and while on average they are very short in terms of duration, they also rely on a resource type which is scarce. This combination could result in a bottleneck to the performance of the entire system, limiting the effectiveness of job dispatching heuristics. The danger of resource fragmentation for GPUs in the system is always around the corner as well. Finally, as these jobs are on average very short, they will also be sensible to the waiting time, which could result in very high slowdown values. Because of this, we will need to pay extreme caution to GPU-Based jobs, and for fairness to all accelerator-based jobs in Eurora.

3.3 Estimating the Job Duration

There are three variants of the Eurora workload, according the estimated job duration values J_{d_e} being used. These are:

- Using the *wall time* J_{d_w} as a job duration estimation; the worst results are expected from this variant, due to its severe over-estimation;
- Using the *real time* J_{d_r} as a job duration estimation; we expect the best results from this variant, which serves mostly as a theoretical *baseline*;
- Using a data-driven *estimation* J_{d_e} computed from the workload.

The last approach we introduced, which uses an estimation computed from the data, is based on a simple heuristic [26]. This technique tries to build *job profiles*, starting from the available log data and user histories: these profiles include the user name, the job name, the queue name, the wall-time, and also the amount of resources that are needed by the job. In simple words, the algorithm will search for past jobs that are similar to the current one, and pick their duration as an estimation. It has in fact been observed that jobs with similar profiles have approximately the same duration for long periods of time: in other words, there is a *temporal locality* principle at work with such jobs. This is due to the fact that, in a certain time frame, the same job might be repeated many times for debugging purposes, or with different parameter combinations, usually employing the same input data. After a certain time, the same profile usually shifts to a new duration, and remains again stable for some time.

The heuristic can be formulated through a set of matching rules with decreasing priority. These are the following, starting from the most important:

1. A full profile match for the job is searched in the user history;
2. A match is searched, allowing the job names to have just a prefix in common. Users, in fact, often name similar jobs incrementally;
3. A match is searched, allowing only the resources use to differ;

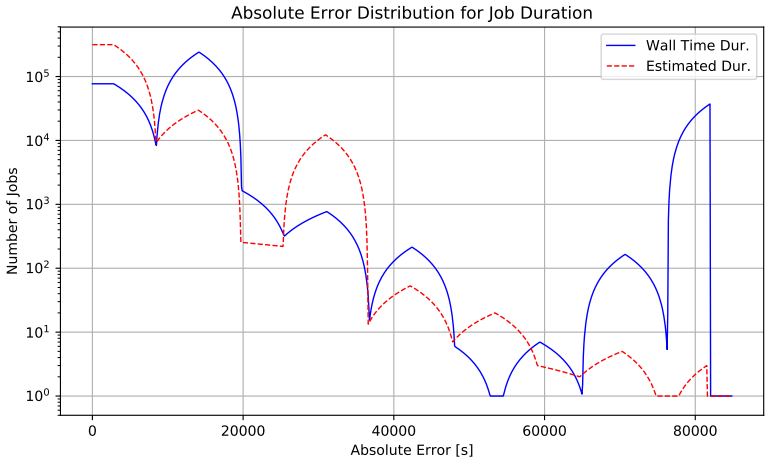


Figure 3.6: Distributions for absolute error in the estimation of job duration, for wall time values and the prediction heuristic, on the Eurora workload.

4. A match is searched, allowing not only the resources use to differ, but also the job name like in rule No.2;
5. A match is searched by using only the job name, or at least its prefix;
6. The job's wall time is used as an estimation.

In Figure 3.6 we can see the absolute error distributions in the estimation of job duration, by using the default wall time values and the prediction method described here. The prediction heuristic fares much better than the wall time: the former achieves an average error of 40 minutes, while the latter has one of 310 minutes. However, unlike the wall-time, our method can under-estimate the durations as well, which must be taken into account.

Chapter 4

The AccaSim Simulator

In this Chapter we will introduce the HPC simulation environment that was developed and used in the scope of the thesis, named AccaSim. We will also discuss the implications of using a simulation environment for HPC dispatching research, and what is the state of the art in this field.

In Section 4.1 we will present the reasons for which the need for HPC simulation systems has arisen. In Section 4.2 we will instead describe the state of the art in this field. In Section 4.3 we will then introduce the AccaSim simulator together with, in Section 4.4, some scalability and performance results obtained with it.

4.1 The Purpose of HPC System Simulators

One may wonder on why an HPC system simulator would be necessary. There are many reasons that have triggered the development of such simulators, among which is the necessity to test and evaluate dispatching methods and system management policies, in the scope of HPC dispatching research.
As we explained in Chapter 2, good dispatching methods can effectively improve in a substantial manner the efficiency and QoS of HPC systems; however, the effectiveness of said methods cannot be estimated *a priori*, but rather depends on many real-world variables, such as the structure of the

workloads or of the system itself. For these reasons, dispatching methods must be thoroughly tested before being used, and evaluated according to the criteria and techniques we described earlier.

Still, this kind of testing cannot be done on a real HPC system, for a wide variety of reasons: first of all, HPC systems usually deliver vital services to users, so they cannot be arbitrarily put under *testing*, thus potentially reducing the quality of service or even disrupting the system's functions. Besides, obtaining a real HPC system may be impossible for researchers. Also, testing should be done on sufficiently big workloads, composed of thousands jobs, in order to obtain meaningful data: this, in a real system, would take *years* of time, and it is, of course, unacceptable. Lastly, one vital condition in order to obtain meaningful and comparable results, is the *repeatability* of experiments: this condition does not apply on real HPC systems, as users freely submit jobs and working conditions change over time; to obtain repeatable experiments, a static workload must be used.
For the reasons above, in order to perform testing of dispatching methods and evaluate them, we need an HPC system simulator, which should be able to replicate the behavior of a real system, while retaining *short* simulation times, and *repeatability* through the use of pre-defined workloads.

4.2 State of the Art

Different kinds of simulators for HPC systems have appeared in the last years, focused on different aspects. For example, in [27] a simulator was developed for the analysis of network systems and topologies in HPC installations; in [28], instead, an HPC system simulator was developed with energy-aware capabilities, named **Performance and Energy-Aware Scheduling** (PEAS).

In our work, we are more interested in simulators focused on the *Workload Management System* (WMS) part, which includes the dispatcher component, and that can be thus used to evaluate different scheduling and allocation heuristics. In this class of simulators, the most recent appears to be

Scheduling Simulation Framework (ScSF) [29]: this simulator is based on an actual installation of the Slurm WMS, which interacts with a virtualized resource layer, and integrates a synthetic job generation system. Being based on a real WMS, ScSF is able to produce very reliable results, that are extremely close to those one would obtain in a real system. However, due to its nature, ScSF is a very resource-hungry simulator, and cannot be used effectively on a commodity machine. For the same reasons, ScSF is hard to setup and not easy to customize.

Among the other notable simulators there is **Cluster Discrete Event Simulator** (CDES) [30], which however is limited to pre-defined scheduling and allocation algorithms, and relies on fixed resource types. CDES, besides, cannot be customized, as its implementation is not publicly available.
Lastly, in [31] an HPC system simulator was developed, based on the Omnet++ software [32]. Such software is however mostly devoted at network simulation and analysis, so the resulting simulator is very limited and difficult to customize, while being also limited to fixed resource types, like CDES.

As it can be seen, the available simulators are mostly targeted at specific purposes and aspects of HPC systems, being in general resource-hungry and difficult to customize.

4.3 Overview of AccaSim

AccaSim is a free, open-source HPC system simulator focused on the evaluation of dispatching methods [6]. It was developed at the *University of Bologna*, Italy, at the Department of Computer Science and Engineering (DISI), and its main contributors are Cristian Galleguillos, Zeynep Kiziltan and Alessio Netti. AccaSim is structured as a *discrete event simulator*, and the simulated system's status is changed according to events that are triggered by the simulator. Since we are simulating HPC systems, these events are mainly represented by the submission, start and termination of jobs.

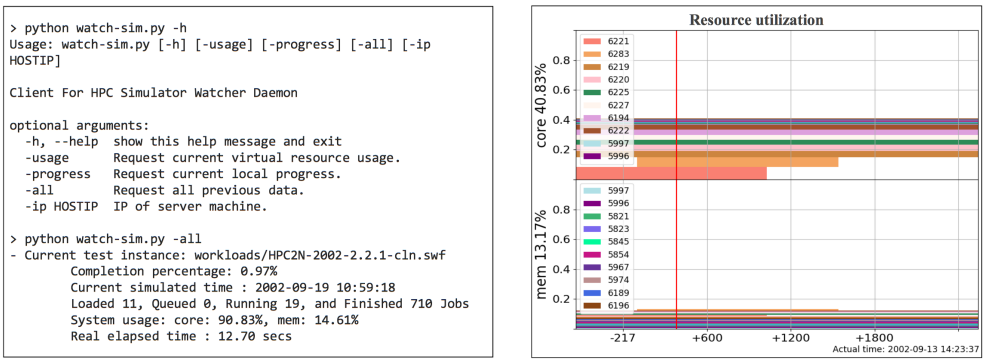
The simulator is written in Python, a high-level, object-oriented, interpreted programming language, and is designed to run with versions 3.4 and above. AccaSim has very few dependencies that are outside of the standard Python distribution, and these are the *psutil*, *json* and *matplotlib* libraries, which can be easily obtained. The simulator's source code is freely available on GitHub [33], and a stable version can be found on the *Pypi* repository.

4.3.1 Main Features

AccaSim is designed to be as flexible and customizable as possible. It allows the definition of arbitrary HPC systems, which can be either homogeneous or heterogeneous. This definition is contained in a *json* configuration file, which is loaded when the simulator is started. AccaSim also allows the use of any kind of workload, as long as it is compatible with the simulated system. The user must supply a parser for the workload's format, in order to correctly read and use it. By default, AccaSim supplies a parser for the **Standard Workload Format** (SWF) [34].

The development of custom dispatching methods is supported as well: users just need to design classes that comply to the default interface used by the simulator for dispatching. At the same time, the simulator can be tuned and improved thanks to its modular structure, making it easy to integrate power-aware or topology-aware behavior. The addition of such secondary information is natively supported by AccaSim, thanks to its *additional data* functionality, which does not imply the modification of the simulator's core. Unfortunately, AccaSim doesn't support multiple job queues yet, meaning that jobs from different queues in such a system will be mixed in one only queue. However, the original information is still present and can be exploited by the dispatcher.

AccaSim also supports various output and monitoring functionalities. First of all, the output of each simulation is given by a *scheduling* file, which will contain records for each dispatched job in the workload, and will be



(a) The Watcher tool. (b) The Visualization tool.

Figure 4.1: The monitoring tools offered by AccaSim. Image taken from [6].

in a user-specified format; the simulator will also output a *pretty print* file, which is the equivalent of the scheduling file, but in a more humanly readable format, and a *statistics* file, which contains several statistics regarding the entire simulation. However, AccaSim also supports the output of detailed *resource usage* logs for benchmarking, with entries containing data regarding the management and dispatching phases in each step of the simulation.

Lastly, in Figure 4.1 we can see some of the monitoring tools offered by AccaSim. The first of these is a *watcher daemon*, which can be remotely queried, and returns information regarding the ongoing simulations. There is also a *visualization tool*, which allows to see the real-time resource usage of the virtual HPC system, mainly for debug purposes.

AccaSim is also designed to be lightweight and scalable: it can be used on any commodity machine, and can be configured in few minutes.

4.3.2 Architecture Overview

In Figure 4.2 we can see a representation of AccaSim's software architecture, which we will now analyze. First of all, we have the *job submission* component, which is tasked with mimicking the submission of jobs by users: a *reader* object will directly read entries from a real or synthetic workload,

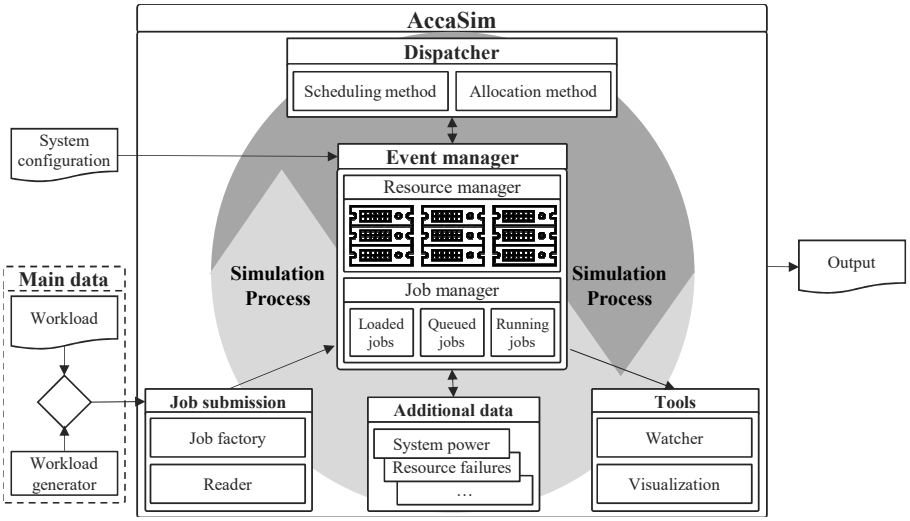


Figure 4.2: AccaSim's architecture. Image taken from [6].

while the *job factory* component will convert these to internal object representations for jobs, ready to be used in the simulator. A *parser* object can be set for use by the reader, in order to add compatibility to different workload formats.

The job submission process itself is handled by the *event manager* component, to which the generated jobs are passed. Such component is tasked with handling the simulated behavior of both the resources in the virtual HPC system, and of the jobs running on it. The *resource manager* and *job manager* sub-components do this, respectively: the two can be used to retrieve information about the system as well, during dispatching.

Finally, the *dispatcher* component in the system includes the *scheduler* and *allocator* sub-components. At each time step, the simulator will invoke the scheduler, which will invoke in turn the allocator for the subset of jobs it has selected from the queue, and that are to be started immediately.

There are various additional components in the simulator's architecture: AccaSim provides an *additional data* interface, allowing to plug secondary information to the simulator, that can then be used by the dispatcher. Such mechanism is transparent, and allows to extend the simulator without ex-

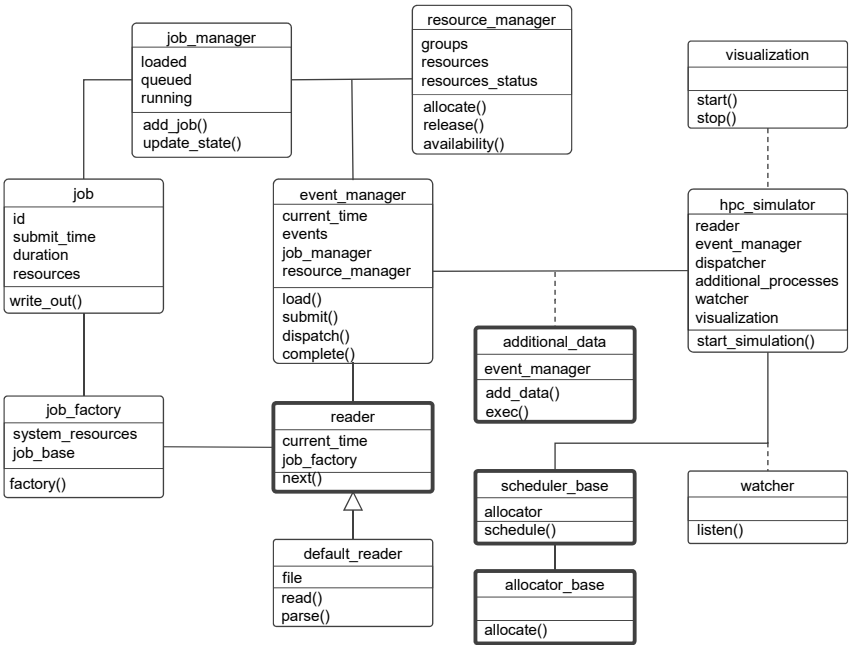


Figure 4.3: AccaSim's class diagram. Image taken from [6].

plicitly modifying the event manager component. Such additional data could be power-related, or even failure-related.

The *tools* part, instead, is represented by all the *daemons* and accessory functions that are user-specified and bound to the simulation, usually being output-related: by default, the *watcher* and *visualization* tools presented earlier are implemented, which are designed to run on separate threads in respect to the simulation.

4.3.3 Implementation

Class Diagram

In Figure 4.3 we can see the UML *class diagram* for AccaSim's core components. While the entire simulator is open and completely modifiable, some specific parts were made to be more easily customizable and interchangeable: these are marked in bold, in the diagram.

Among the customizable parts, we can see the scheduler and allocator compo-

nents, represented by the *scheduler_base* and *allocator_base* abstract classes: users can design custom dispatching methods, just by creating new classes that implement these two interfaces. Along the same lines, users can specify custom reader objects, by implementing the *reader* abstract class, as well as *parser* objects, adding compatibility to diverse workload formats. The default reader, predictably named *default_reader*, supports the SWF format. Finally, by extending the *additional_data* class, users can supply additional information which can be used by the simulator in a transparent manner.

The Scheduler and Allocator Interfaces

We will dedicate some time to how the *scheduler_base* and *allocator_base* interfaces are structured, since such entities are so important in our work. Starting from the *scheduler_base* interface, it is designed so that the simulator can invoke the scheduler through a standard *schedule* method. Its syntax is the following:

```
schedule(cur_time, es_dict, es, debug)
```

The *cur_time* argument simply indicates the current time; the *es_dict* argument, instead, points to the job dictionary used by the simulator, in order to retrieve information about single jobs by using the corresponding IDs; the *es* argument is the job queue itself, or a subset of it, which is structured as a list of IDs. The *debug* argument, finally, is just a boolean flag allowing to turn on and off verbose output. The method must return an array of tuples, one for each job in *es*, each containing respectively the starting time (if found), the job's ID, and a list of length J_n defining to which node each job unit was assigned. The simulator will then proceed to dispatch all jobs that have a starting time corresponding to the current time.

Proceeding to the *allocator_base* interface, it is designed so that it can be used by any scheduler in a transparent way, through an *allocate* method. Its syntax is the following:

```
allocate(es, cur_time, skip, reserved_time, reserved_nodes,
    ↪ debug)
```

This method's arguments are similar to the ones specified earlier. Here however, the *es* list contains references to the job objects themselves, in this case the ones supplied by the scheduler and for which an allocation is to be computed. The *skip* parameter is a boolean, which can allow the allocator to skip jobs in *es* if an allocation cannot be found, instead of stopping the allocation process altogether. This can be needed by some schedulers. The *reserved_time* and *reserved_nodes* arguments, instead, are related to backfill implementations, and allow to set a list of reserved nodes that are to be avoided by the allocator in a certain time frame. The method has the same return type as the *schedule* one. Depending on wether the allocations were successful or not, the scheduler will then decide how to proceed.

Simulation Process

We will now describe the simulator's behavior at runtime. First of all, an *hpc_simulator* object must be instanced. Its constructor will take the following arguments:

- The filepath of the workload to be used;
- The filepath of the system configuration file;
- A scheduler object.

The simulator's constructor also accepts an optional *reader* object, for workloads written in custom format. If none is specified, a *default_reader* instance will be used, again for the SWF format. The system configuration file, written in json, will contain the simulator's settings, in particular regarding the output format, and will also contain the path to a second configuration file, which will instead define the structure of the simulated HPC system: such file will be divided in two parts, the first of which will detail the node types in the system, in terms of resource availability, while the second

part will define how many nodes are available for each type. The scheduler object, instead, must be an instance of a *scheduler_base*-like class, which in its constructor needs in turn an *allocator_base*-like object, thus defining the complete dispatching method.

Having instanced the simulator object, the simulation can be started through the *start_simulation* method, which has the following syntax:

```
start_simulation(visualization, watcher, debug)
```

This method accepts various boolean arguments, allowing to enable or disable *verbose output*, besides the watcher and visualization tools.
AccaSim, as mentioned earlier, is a discrete event simulator, meaning that it will progress through *discrete time steps*: these steps correspond to events that change the status of the system, which in this case are the submission, the start and termination of jobs. Time steps are thus dynamically added as new jobs are loaded and dispatched, and used to advance the simulation.
Jobs themselves can have various statuses. Such statuses are *loaded*, *queued*, *running* and *completed*. A job which has been read from the workload will shift from the *loaded* to the *queued* status, when its submission time J_{t_q} is reached. At this point it will be added to the internal job queue. Once dispatched, the job will then reach the *running* status, and when terminated it will finally reach the *completed* status. The real duration J_{t_r} for each job is known only by the simulator itself, and not by the dispatching component: like in a real system, the latter can only use the estimated duration J_{t_e} and wall time J_{t_w} values in its computations.

The main simulation loop is fairly simple. At every simulation time step, the simulator will execute the following procedures:

1. **release** the resources related to jobs that have terminated in the current time step; the jobs are then removed from all data structures in the simulator, shifted to the *completed* state, and their entries are written to the output scheduling file;

2. if there are pending jobs in the queue, perform **dispatching** by invoking the scheduler, which will pick a set of jobs from the queue, and pass them to the allocator; the scheduler will then return the set of jobs that can be dispatched *immediately*, together with their resource assignments;

3. if there are jobs to be dispatched, **start** them on the virtual system through the resource manager component, thus modifying their state from *queued* to *running*; new time steps corresponding to the real and estimated termination times for all started jobs are also added, by using their J_{t_r} and J_{t_e} values;

4. if the amount of jobs that are loaded but not yet queued is below a certain threshold, **load** the next jobs from the workload, and add the corresponding submission time steps; these jobs will be in the *loaded* state. The reading process is incremental, in order to minimize the memory usage related to big workloads;

5. **transfer** on the queue all jobs submitted at the next time step, shifting their state from *loaded* to *queued*, and set this one step as current;

6. if its output is enabled, a new entry is **written** to the resource usage log file, detailing how much time was taken by the simulation and dispatching phases for the current time step, besides memory usage and queue size.

The simulation ends when there are no more *queued*, *running* or *loaded* jobs, and the selected workload file has been completely read. At this point the output statistics file is written, detailing the total simulation time, the average waiting and slowdown times, and other such metrics. The simulator will then terminate.

4.4 Performance of the Simulator

In this section we will evaluate the performance of the simulator, after describing which tests were performed and on which data, in order to support the claim that AccaSim is fast and lightweight.

4.4.1 Test Methodology

In order to perform our tests, we chose to model the **Seth** system [35], located at the *High-Performance Computing Center North* (HPC2N), in Sweden. The system, built in 2001 and now retired, is composed of 120 nodes, each having two AMD Athlon MP2000+ dual-core CPUs and 1GB of RAM. There is a public workload available for the Seth system extracted from its log trace [36], including roughly 200000 jobs and spanning through 3.5 years, from July 2002 to January 2006. The workload is in SWF format, and thus compatible with the simulator's default parser. The configuration files, together with the workload we used, are available as examples on AccaSim's GitHub repository.

Said workload was then tested with a series of dispatching methods. We employed four different scheduler types, namely First-Come First-Served (here named FIFO according to its implementation), Longest Job First (LJF), Shortest Job First (SJF) and Easy Backfill (EBF). With these schedulers, two different allocation heuristics were used, which were First Fit (FF) and Best Fit (C), resulting in a total of eight combinations. The eight tests, split in sets of four on two threads, were performed on an Apple Macbook Pro Machine, mounting an Intel dual-core i5@2.2Ghz CPU, and 8GB of RAM.

4.4.2 Performance Results

We will now discuss the performance of the simulator from diverse points of view. All the data and plots proposed here were computed starting from the resource usage logs written for each test by AccaSim. In Table 4.1 we can see various resource usage statistics for the test instances; first of all,

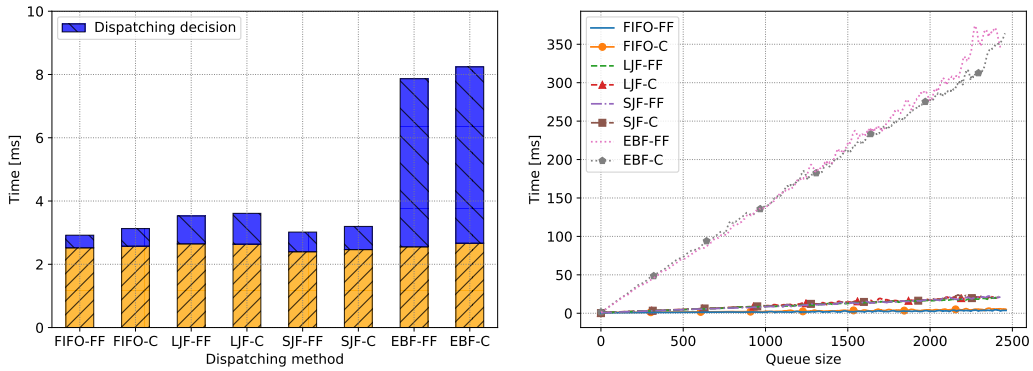
Disp. Method	**Tot. Time** [mm:ss]	**Disp. Time** [mm:ss]	**Avg. Mem.** [MB]	**Max. Mem.** [MB]
FIFO-FF	24:44	03:23	27.8	41.4
FIFO-C	26:31	04:43	27.5	40.9
LJF-FF	29:45	07:26	32.7	54.7
LJF-C	30:26	08:12	32.9	54.7
SJF-FF	25:48	05:19	32.5	54.7
SJF-C	27:19	06:14	32.5	55.4
EBF-FF	68:27	46:15	26.8	40.2
EBF-C	71:43	48:31	26.9	40.2

Table 4.1: Resource usage of the simulator.

we can see that across all instances RAM usage peaks at 55MB, keeping an average below 30MB: AccaSim's need for so little RAM memory grants it great parallelization potential.

From the CPU point of view, we can see that most simulation instances lasted around 30 minutes, with the only exception being the EBF instances, that lasted roughly 70 minutes. This is to be expected, as Backfill is a much more computationally intensive scheduling algorithm compared to the others. We can also see the amount of simulation time that was specifically reserved to dispatching: once again, we can see higher values for EBF. The remaining time, which is used by the simulator to perform management tasks, such as the loading, starting and termination of jobs, is mostly constant across all test instances, and amounts to roughly 22 minutes. This is, once again, a reasonable value, and the similar behavior of the simulator on all instances makes it a reliable and predictable tool. The full simulation time, obviously, also strongly depends on the specific workload, with higher times related to more difficult and bigger instances.

In Figure 4.4a, we can see the average time required per simulation step on all test instances. The results confirm what we have said earlier: the simulator itself, in its management tasks, takes a low, homogeneous footprint on all instances, averaging slightly above 2ms; the rest of the time, which is related to dispatching, depends on the scheduling and allocation heuristics that were used.



(a) Average CPU time required at each time step.

(b) Average dispatching time in function of the queue size.

Figure 4.4: Scalability of the simulation process. Image taken from [6].

We also propose a *scalability*-oriented analysis: in Figure 4.4b, we can see a plot depicting the time required by a specific dispatching method in function of the queue size, again obtained from the test data, thus estimating the computational complexity of each dispatching method. While all methods retain a linear behavior, we can clearly see that EBF is much more intensive than the others, scaling worse with higher queue sizes. It can also be seen that the allocation heuristic being used does not make a big difference, suggesting that the overall time complexity is dominated by the scheduler.
This kind of plot can be really useful, as it allows to perform a reliable scalability analysis through real data in real-world conditions, proving AccaSim's potential as a testing and evaluation tool for dispatching methods.

These results shown by AccaSim are highly satisfying, as despite the workload's size, the simulator has still shown reasonably low simulation times and a constant, small and reliable footprint on the system's resources.

Chapter 5

Developing New Dispatching Heuristics

In this chapter we will finally face the development of new dispatching heuristics for our target system, which is Eurora: we will thus present the scheduling and allocation heuristics that were actively developed.

In Section 5.1 we will discuss the design and development approach we pursued. In Sections 5.2 and 5.3, we will then discuss the scheduling and allocation heuristics, respectively, that were developed and that will be tested against the available workload.

5.1 Approach and Methodology

While developing new dispatching heuristics, we will adopt a *data-driven* approach: the design and implementation process will not be guided by theoretical or intuitive assumptions, but rather by practical analysis on the available data, which will provide us with a robust knowledge and understanding about the system's behavior. At the same time, constant testing will have a big influence on the development process, thus allowing us to improve our heuristics as we are able to see how they behave on our system. This kind of approach can be adopted, of course, because we have the right

instruments for it: namely, a large and reliable workload, and an efficient simulation environment. Our approach is data-driven also in other ways: some of the heuristics we will use, for example to predict the job duration as described in Section 3.3, are deeply data-driven as well, in the sense that they employ data matching techniques rather than ones belonging to traditional programming.

We should also point out what was effectively done in the scope of the thesis: the scheduling algorithms that will be presented were already available, and on these the work was focused on improvement. The biggest part of the work was done on the allocation heuristics, which were designed and implemented from scratch. A specific framework was designed as well: such framework is the one resulting in AccaSim, presented in Chapter 4, which allows the integration between schedulers and allocators.

5.2 Available Scheduling Algorithms

In this section we will present and describe the scheduling methods, which were already available for use, and that will be used for testing. All of the following algorithms were implemented according to the *scheduler_base* interface for AccaSim described in Section 4.3.3.

5.2.1 Simple Heuristic

The first scheduling algorithm we will propose is a simple queueing-oriented algorithm, which just sorts the job queue according to a specific criteria. This method is implemented in the *simple_heuristic* class, and can operate in three different modes. These are *First Come First Served* (FCFS), *Shortest Job First* (SJF) and *Longest Job First* (LJF). These three modes are implemented, obviously, by using the estimated duration value J_{d_e} for each job in the queue.

The algorithm's behavior is very simple: at each invocation it will sort the

job queue passed to it by the WMS, and then try to schedule each single job to start immediately. In order to do so, each job is passed to the allocator, which will return a suitable resource assignment for it. When an allocation fails, and a job cannot be scheduled, the scheduler will stop and return the list of jobs that were successfully allocated and that can be started immediately. This is the implementation that was used in Section 4.4 to test the AccaSim simulator with the FCFS, SJF and LJF algorithms.

5.2.2 Easy Backfill

The second scheduling algorithm that was implemented, still in a queueing-oriented fashion, is an Easy Backfill algorithm. Its implementation can be found in the *backfill_heuristic* class.

This Easy Backfill implementation does not perform any kind of sorting on the job queue, which is left in its FIFO ordering. It should be pointed out, however, that at this stage the reservations for blocked jobs are managed on a *per-node* basis: this means that a whole node is always reserved for a blocked job, and not a subset of its resources. When the reservation time is reached, the allocator will then pick the subset of resources actually needed by the job. The algorithm could thus be improved by allowing other jobs to be allocated on the reserved nodes, in backfill mode, as long as they leave an amount of resources which is sufficient for the reservation. However, we are not sure if this could bring any sensible benefit. Since this method strictly follows the behavior of the algorithm found in literature, there is not much left to say about it. This is also the implementation used in Section 4.4 to test the AccaSim simulator.

5.2.3 Priority Rule-Based

The heuristic we will present here is again queueing-oriented, and of the priority rule-based type [7]. It is implemented in the *prb_heuristic* class.

This PRB algorithm is functionally very similar to the previous Simple

Heuristic we presented. However, in this case the jobs are ranked according to their estimated *tardiness*, first introduced in Section 2.1.3, which is the relative delay compared to the expected waiting time for the job's queue. As presented in Section 3.1, in fact, the Eurora system has three job queues, each with a different estimated waiting time depending on its specific purpose. This value, defined as ewt_q, is assumed to be of 1 hour for the *debug* queue, of 6 hours for the *parallel* queue, and finally of 24 hours for the *longpar* queue. These are taken into account by the scheduler, as AccaSim does not natively support multiple job queues. At this point, to each job can be associated the following ranking:

$$rank_J = -\frac{max(ewt) * wait_J}{ewt_{J_q}} \tag{5.1}$$

In Equation 5.1, $wait_J$ represents the current waiting time for job J, while ewt_{J_q} is the expected waiting time for the queue it belongs to. $max(ewt)$, finally, is simply a normalization factor which corresponds to the maximum ewt value for all queues in the system. This way, the algorithm will always pick the jobs with the greatest tardiness value in the queue first.
There is also a tie-breaker mechanism in place, for jobs that share the same ranking. This secondary ranking is given by the following formula:

$$tb_J = J_{d_e} * [J_n * \sum_{k \in res} J_{r,k}] \tag{5.2}$$

In Equation 5.2, J_{d_e} is the expected duration for job J, while J_n is the number of requested job units, and $J_{r,k}$ is the resource request for a specific resource type k in the system, in a single job unit. The product seen in the second member simply expresses the total amount of resources requested by the job. This amount, multiplied by the job's expected duration, is usually referred to as the job's *geometry*, which is a simple approximation of how much a certain job is going to keep the system busy. In this case, the tie-breaker mechanism will favor jobs with a smaller geometry, which require less resources and last for a lower time.

It should be noted that this PRB algorithm, unlike the Simple Heuristic, will not stop at the first allocation fail: instead, it will still try to allocate the remaining jobs in the queue, until the end of it is reached. This can help to improve the throughput, but could introduce the risk of starvation for some jobs.

5.2.4 Constraint Programming-Based

The scheduling algorithm we will now present is planning-oriented, unlike the ones we have discussed before. It is implemented in the *cpa_scheduler* class, and is based on the *constraint programming* technique, which is an highly optimized form of heuristic search based on sets of constraints that are to be respected, and used to prune the search tree [7]. The algorithm, specifically, employs the **or-tools** library made by Google [37], which implements various models and search algorithms for constraint programming and optimization in general.

As previously explained, a planning-oriented scheduling algorithm will try to generate a schedule plan, thus assigning certain starting times to all jobs in the queue. In this case, the size of the problem is bounded, in order to cope with the online nature of the underlying system: every time the algorithm is invoked, the job queue is sorted in the same way as in the Priority Rule-Based algorithm presented earlier. Out of this sorted queue, a feasible schedule will be searched only for the first N jobs at most (by default 100). The remaining jobs won't be considered in the optimization process, and will be statically scheduled sequentially, one after the other, beyond a *maximum makespan* point. This maximum makespan point is defined as follows:

$$max\_mks = t + \sum_{J^i \in Q} J^i_{d_e} + \sum_{J^i \in R} [J^i_{d_e} - (t - J^i_{t_s})] \tag{5.3}$$

In Equation 5.3, t represents the current time, while R is the set of running jobs, and Q is the job queue. Basically, we are summing the total estimated duration of all jobs in the queue, and the total estimated remaining time for

running jobs. This, in other words, is a worst-case makespan, obtained if all jobs are scheduled and complete in a sequential manner. This value is also an upper bound to the search space for the jobs' starting times, which is limited to the interval $[t, max\_mks]$.

The set of **decision variables** used in the model is constituted of *Interval* variables, one for each job in the subset of the queue being considered. An Interval variable specifies the starting point J_{t_s} of an activity, which is characterized by a fixed duration, in this case J_{d_e}. Such starting point has also lower and upper bounds associated to it for the search, which are once again t and max_mks, making up the variables' domains of admissible values as well.

The model is then characterized by a set of **constraints**, which are of the *Cumulative* type, and are one for each resource type k in the system. Thus, for Eurora, we will have four such constraints, namely for *core*, *memory*, *gpu* and *mic* resources. Cumulative, which is a *global* constraint, is formulated as follows:

$$\begin{aligned} &\text{cumulative}([S^1, S^2, ..., S^n], [D^1, D^2, ..., D^n], [C^1, C^2, ..., C^n], C) \\ &\qquad \text{iff} \quad \sum_{i|S^i \leq u < S^i + D^i} C^i \leq C \quad \forall u \in D \end{aligned} \tag{5.4}$$

In Equation 5.4 we are imposing that all tasks i, with certain starting times S^i, durations D^i, and resource requests C^i, must never violate the total capacity C for such resource, at any time u. D is, finally, the domain of the S variables. In simple words, jobs must never be scheduled so that they would use more resources than those available in the system. In our specific case, the parameters for the equations above would be the following:

$$\begin{aligned} S^i &= J^i_{t_s} \\ D^i &= J^i_{d_e} \\ C &= C_k \qquad \forall 1 \leq i \leq n \\ C^i &= J^i_n * J^i_{r,k} \\ D &= [t, max\_mks] \end{aligned} \tag{5.5}$$

In Equation 5.5, the quantity $J^i_n * J^i_{r,k}$ expresses the total request for resource type k by job J^i, while C_k is the total capacity for such resource type in the system. The jobs J^i being considered for these constraints are those in the queue to be scheduled, and those that are currently running as well. Since we are only considering the *total* resource availability for each type in the system, without distinguishing between the single nodes, and we are using the estimated duration for each job, the resulting model is *relaxed*. This means that some inconsistencies could be found in the resulting schedule plans, which will then be detected by the allocator, generating a *feedback loop* between the two entities.

Finally, the **objective function** that will be minimized is the average *normalized tardiness* seen in the Priority Rule-Based scheduler, as its first ranking operator $rank_J$. The resulting formula is the following:

$$f(S) = \frac{1}{N} \sum_{J \in S} - \frac{max(ewt) * wait_J}{ewt_{J_q}} \tag{5.6}$$

In Equation 5.6, S represents a full or partial solution that was found, containing the assignments for the jobs' starting times. This is the function that guides the branching and variable choice strategies as well.

At this point, the search procedure itself can be started: given the variables to assign, the constraints to respect and the objective function to minimize, an heuristic search is performed with the *CPSolver* or-tools class. This search process is *time-limited*: if the algorithm cannot complete within the basic time limit (1 second) and no solution is found, a new search is started

by doubling such time limit. The process is repeated until a solution is found, or a maximum time limit is reached, in which case no job is scheduled. In general though, to higher time limits correspond better solutions. In this case we are obviously not interested in obtaining the optimal solution, but rather we prefer sub-optimal ones that work just as well, and that can be obtained in a reasonable time.
After a schedule plan is found, all jobs with a starting point corresponding to the current time are supplied to the chosen allocator: in this phase, it could happen that some inconsistencies are found by it, and that some of the jobs cannot be dispatched. These jobs are left in the queue, and will be again considered for scheduling in the future, when a new search will be performed. The other jobs, scheduled to start later, are simply discarded, and will be considered again at the next scheduling call for a new search. This is the easiest solution to make the algorithm adaptable against new jobs arriving in the queue, job duration over/under-estimations, and allocation inconsistencies.

5.3 Developed Allocation Heuristics

In this section we will now finally discuss the active development of allocation heuristics, done in the scope of the thesis. The algorithms are all based on the *allocator_base* interface in AccaSim, introduced in Section 4.3.3, and all belong to the first-fit type of heuristics.

5.3.1 First-Fit Heuristic

The first allocation heuristic we will present is a simple first-fit algorithm, which does not sort nodes in the system. It is implemented in the *allocator_simple* class, which is very important in the project, as it implements the basic allocation loop that will be used by all of the other heuristics. The entire allocator collection is in fact implemented in a object-oriented way, and the other allocators just *extend* the Simple one, in order to maximize code re-use and reliability: the full class diagram can be seen in Figure 5.1.

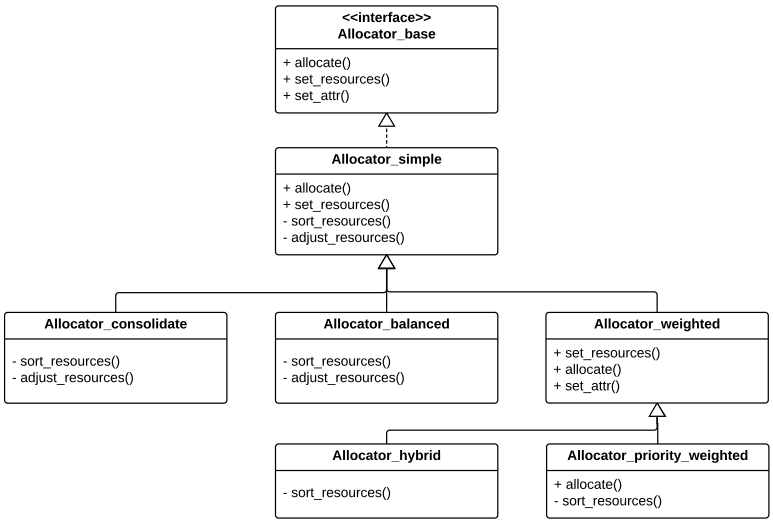


Figure 5.1: The class diagram for the package of allocators that was made.

The algorithm accepts single jobs or lists of jobs indifferently. We will summarize its flow in the set of steps that follows:

1. Prepare the **list of nodes** in the system, through the *sort_resources* method, which returns a list of node IDs. In this specific algorithm the method is a *dummy*, and will just return the list of nodes as given by the resource manager without sorting them in any way;

2. For each job in the input list to be allocated:

   (a) If the *reserved_nodes* and *reserved_time* arguments are present, compute the list of nodes that cannot be used, as they are **reserved** by a backfill algorithm, by checking the overlap between the reservation time and the job's expected duration in respect to the current time;

   (b) Traverse the sorted list of nodes; for each node, compute its *fit*, which is the number of job units fitting in it, and **allocate** as many of them as possible; the cycle continues until all of the job's requested units are allocated, or the end of the list is reached;

(c) If the allocation is **successful**, add its entry to the output list, and sort again the list of nodes for the next job, with the *adjust_resources* method, which in this case is a dummy as well. If the allocation was **unsuccessful**, there are two possibilities: if the *skip* parameter is set to True, the entry for the job's failed allocation is added to the output list and the algorithm continues with the next job; if the *skip* parameter is set to False, instead, all of the subsequent jobs automatically fail as well and all of their entries are written to the output list;

3. When all jobs have been processed return the allocation output list, containing the node assignments for jobs to be dispatched immediately.

The algorithm was built to be as modular and flexible as possible. Since the *sort_resources* and *adjust_resources* methods are already embedded in the main allocation loop, any more elaborate allocator that wants to perform sorting on the system's nodes can be implemented by just extending these two methods, without touching the *allocate* method at all. The distinction between the two methods is also important: *sort_resources* must create from scratch a sorted version of the nodes' list, while *adjust_resources* will start from an almost-sorted version of it resulting from the allocation of a job. This approach leads to a huge performance improvement, as sorting an almost-sorted list is usually very efficient.
The choice of allocating as many job units as possible in a single node is natural as well: this way, we can greatly reduce strain on the network and minimize the risk for resource fragmentation, by performing consolidation. The allocator supports multiple reservations as well, making it compatible with Conservative Backfill algorithms, and increasing its versatility.

The time complexity for this allocation algorithm is estimated to be, in the worst case, at $O(NM)$, where N is the number of jobs and M the number of nodes in the system, assuming a constant, low, number of resource types k. This worst case is reached when none of the jobs in the list can be allocated.

5.3.2 Best-Fit Heuristic

The algorithm we will present here is a best-fit heuristic, and is implemented in the *allocator_consolidate* class.

In this case, the best-fit condition is obtained by summing *all* resources contained in a node, for all types, and by sorting them in ascending order according to such amount. This implementation, in particular, is done by extending the *allocator_simple* class: only the *sort_resources* and *adjust_resources* methods were extended, adding the necessary code to perform sorting according to the best-fit policy.

This algorithm has a slightly higher time complexity compared to the simple first-fit one. The sorting methods use python's *sort* algorithm, which is based on an implementation of *TimSort* [38]. This algorithm has a worst-case performance of $O(nlog(n))$, and a best-case performance of $O(n)$, which is achieved on almost-sorted arrays. We can safely say that the *sort_resources* method will thus have a complexity of $O(Mlog(M))$, while *adjust_resources* will have one of $O(M)$, with M being again the number of nodes in the system. The overall complexity of the algorithm would then be $O(Mlog(M) + MN)$, which can be approximated again to $O(MN)$ if M and N, which is the number of jobs, are in the same order of magnitude.

5.3.3 Balanced Heuristic

The heuristic we will present here is completely different from the ones seen before, and is specifically targeted at Eurora and at other heterogeneous systems as well. It is implemented in the *allocator_balanced* class, and like the best-fit heuristic, it is based on the *simple_allocator* algorithm, with only the sorting methods differing.

We will start from a simple observation made on the Eurora system: as said in Section 3.1, half of the nodes in the system include *GPU* accelerators, while the second half includes *MIC* units. In the default ordering, based on

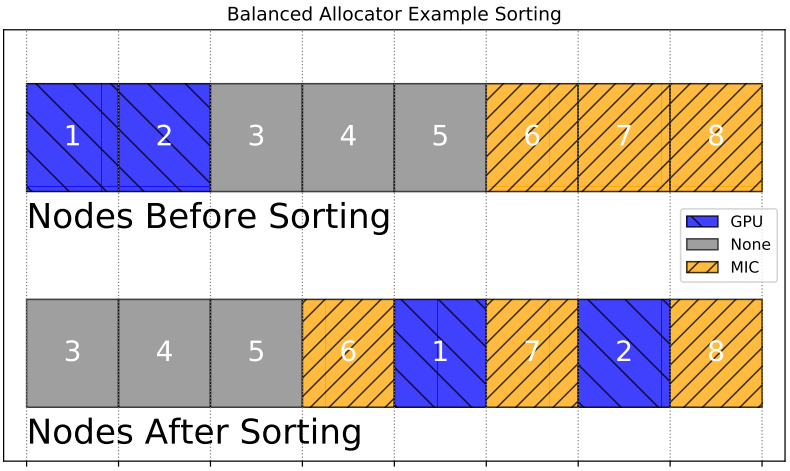


Figure 5.2: An example of the sorting technique used in Balanced on a facsimile of the Eurora system with 8 nodes, the first 4 having GPU resources, and the remaining 4 MIC units.

their IDs, the nodes are divided in two distinct blocks depending on the accelerator unit they have: specifically, we will find GPU units on the first 32 nodes, and MIC units on the last 32. This can lead to a series of problems: when the system is almost empty, or many nodes have the same amount of resources available, any sorting method will depend solely on the node IDs. This means that many jobs will be allocated on nodes that have GPU units, leading to fragmentation for such resource type, and to an unfair disadvantage for jobs requiring them, which are also the most common for this system. Apart from this observation, it would be also natural in a heterogeneous system like Eurora to preserve scarce resources like the accelerator units, by avoiding fragmentation in the respective nodes.

Basing on these preliminary observations, we designed the Balanced allocator, which is made to protect and balance the use of nodes possessing certain resource types. Such balancing effect is achieved, in simple words, by *interleaving* nodes having accelerator-like resources of different types, thus not favoring any of them. This set of *critical resource types*, which the algorithm must consider, can be set through its constructor: by default, these are

the MIC and GPU types. The algorithm itself is rather simple, and works in two phases:

1. All nodes in the system are **collected** in *bins*: there is a bin for each critical resource type, and nodes are assigned to a specific bin according to which of those they currently have available. If they do not have any, they will be assigned to a special *none* bin. It may happen on some systems that a node has multiple critical resource types: in this case, it will be assigned to the bin for which it has the maximum availability;

2. The bins are **combined** in one only node list, which is built as follows: at its head, there will be the nodes belonging to the *none* list, which do not have any critical resources. Then, the rest of the list is built, at each step, by picking a node from the *currently longest* bin, until they are all empty. This approach is very similar to just interleaving the bins, but is more robust when these are of different lengths, as nodes possessing very scarce resource types are selected as last.

The Balanced allocator will avoid the allocation of jobs to nodes possessing critical and scarce resources; even when these critical nodes are selected, the ordering is such that no specific type of resource is penalized, but jobs are rather *distributed* on nodes having different types of critical resources. An example of which are the actual results of this sorting technique can be seen in Figure 5.2. This allocator is most effective when the system is in a low resource utilization state, and thus its balancing feature is more critical. It should be noted, however, that it is designed with many assumptions in mind, and for a specific system: it would be useless for homogeneous systems, and even on other heterogeneous systems there would be no guarantee of its effectiveness.

The algorithm is rather efficient: by default, no sorting is performed, and since the first list-building phase of the algorithm has a linear cost of $O(M)$, the resulting complexity is equal to $O(NM)$, exactly like in the first-fit implementation.

5.3.4 Weighted Heuristic

The Weighted heuristic is an improved version of the best-fit one, and is implemented in the *allocator_weighted* class, which is also an extension of *simple_allocator*.

Weighted performs sorting on the nodes' list specifically for each job from scratch, and does not adjust the sorted list after each allocation. Standard best-fit performs in fact a job-independent sorting, and does not consider the *impact* a job's allocation has on a node, thus leading to potential resource waste. Weighted does fix this: for each node, the number of fitting job units is computed, and the amount of resources that would be left in the node after such allocation is used to compute its ranking. Such sum is weighted for each resource type as follows:

$$\begin{aligned} \overline{req_k} &= \frac{\sum_{J \in Q} J_{d_e} * J_n * j_{r,k}}{\sum_{J \in Q} J_{d_e}} \\ w_k &= \frac{\overline{req_k} * load_k}{avl_k} \end{aligned} \tag{5.7}$$

In Equation 5.7, w_k is the weight itself which will be associated to resource type k, while $\overline{req_k}$ represents the *average request* for such resource type by jobs J in the queue Q. This average is weighted by the expected jobs' duration, thus considering their geometry as well. $load_k$ is instead the *load ratio* for resource k, and avl_k is its total *base availability* in the system, acting as a normalization factor.
In other words, resources are weighted depending on how much they are needed by jobs in the queue, and on their availability in the system: nodes with scarcer, more needed resources will be put towards the end of the list in order to be protected, similarly to what Balanced does with GPU and MIC resources. The presence of complementary heuristic factors such as $load_k$ and $\overline{req_k}$ allows Weighted to perform well in most scenarios, for example when the system has a small resource load or there are too few jobs in the queue

to be considered. Many other solutions were tried, by removing components in the heuristics or employing variants of them, but the one presented here emerged to be the best-performing one.
Additionally, for schedulers like the Constraint Programming-Based one, which compute actual schedule plans, the allocator is able to consider all jobs scheduled in the future that are expected to overlap with the current one, as well.

To limit the cost of the algorithm not all jobs in the queue are considered, but rather a sliding window of fixed length is used, starting from the current job; besides, nodes that do not fit the considered job are discarded before the sorting stage, thus improving its efficiency. Still, the algorithm is more expensive than previous allocators, with a worst-case complexity of $O(NMlog(M))$, since sorting has to be performed for each job.

Weighted is an effective and flexible allocator, often with performance similar to Balanced, but without its strict design assumptions. It is, however, less performant than the latter when the system is in a low resource utilization state, as it does not possess its interleaving capability.

5.3.5 Hybridization Strategies

The Weighted and Balanced allocators have both good performance, but in different scenarios, with Balanced usually performing better than Weighted when the system has low resource load and vice versa. We will now present some *hybridization* strategies that were developed, which try to combine the strong points of both allocators in order to obtain optimal performance.

Hybrid Heuristic

The *Hybrid* allocator, implemented in the *allocator_hybrid* class, is literally a combination of the Weighted and Balanced allocators. Specifically, it is an extension of the *allocator_weighted* class, but accepts in its constructor a list of critical resource types like Balanced.

In this case, sorting on the nodes' list is still performed per-job like in Weighted, but the sorting procedure itself integrates the reasoning seen in Balanced: the nodes are collected in bins like we have seen earlier, depending on the set of critical resource types, but before these bins are combined in the final list, they are sorted one by one according to the Weighted heuristic. This allows to preserve the improved best-fit ranking seen in Weighted, while integrating also the balancing and protection feature for critical resources seen in Balanced, which cannot be achieved through standard sorting.

Hybrid is a *worst-case* approach to the critical resources problem presented earlier, and assumes fragmentation for those resources must be avoided at all cost, at all times, without making assumptions on the workload's distribution. In fact, jobs needing such resources might be very rarely submitted, and since the ordering imposed by the Balanced component is dominant over the Weighted one, which is limited to the single bins, sub-optimal allocation decisions may be performed. The complexity of the algorithm is dominated by the Weighted sorting technique, and thus also results to be $O(NMlog(M))$.

Priority-Weighted Heuristic

Priority-Weighted is the second hybridization strategy we will propose. It is implemented in the *allocator_priority_weighted* class, which is again an extension of *allocator_weighted*, and accepts a set of critical resource types in its constructor.

This heuristic is a *relaxed* version of the Hybrid one, and is designed with the purpose of protecting certain critical resource types, only if they are actually needed by jobs, thus without using Balanced's approach. This is achieved by adding a new component to Weighted's heuristic for resource ranking, as in the following equation:

$$w_k = \frac{\overline{req_k} * load_k * p_k}{avl_k} \tag{5.8}$$

In Equation 5.8, p_k acts as a *priority value* for each critical resource type that was specified. For the other resource types, it is assumed to be always equal to 1. Such priority value is implemented in a very simple way: every time the allocation for a job requiring a certain critical resource k fails, the priority value p_k for such resource type is increased by a unitary value. Conversely, when an allocation is successful, the corresponding value p_k is decreased. If a job requires multiple critical resource types, all of their priority values will be affected. The range in which the priority values can change, together with the increment step, can be user-specified in the allocator constructor, and are by default set respectively to $[1, 10]$ and 1.

This solution allows the allocator to dynamically react to new jobs arriving in the queue, thus taking into account the distribution of the workload, and not treating all resource types equally. At the same time, since this approach is seamlessly integrated in the Weighted heuristic, the allocator will never pick highly ineffective allocation decisions: a node having critical resources available, but with a good fit, will always be picked over a node which has no critical resources available and a much worse fit. This condition does not apply in Hybrid, which forces Balanced's reasoning on Weighted.
Various other solutions were also tried for p_k: for example, we tried using as priority values the average number of allocation failures per-job or per-step, the number of jobs in the queue for which allocation has failed, or even some time-cooling priority mechanisms. Out of all of these, our priority mechanism emerged to be the best technique, despite its simplicity.

Priority-Weighted is not always better than Hybrid: the two algorithms have different strengths, with Priority-Weighted usually picking better allocation decisions and being more adaptable to the workload distribution, and Hybrid being able to better manage the system when it is not much loaded. We must remember, in fact, that Hybrid implements the interleaving technique seen in Balanced, while Priority-Weighted does not. The algorithm has the same time complexity of Weighted, at $O(NMlog(M))$, since it is a natural extension of it.

Chapter 6

Experimental Results

In this chapter we will finally look at the experimental results obtained with the scheduling and allocation heuristics described earlier, on the Eurora workload. We will then draw our conclusions, after having described the behavior of the system.

In Section 6.1 we will describe the test methodology, together with the evaluation metrics that will be used. In Section 6.2 we will then look at the results obtained with the full Eurora workload and in Section 6.3, instead, we will analyze some more specific, focused test cases. In Section 6.4, finally, we will draw our conclusions on the Eurora system and on the developed heuristics.

6.1 Test Methodology

A wide number of tests was performed in the scope of the thesis. We used the three variants of the Eurora workload, corresponding to the wall time, real and estimated durations for jobs. Each of these three variants was tested against all of the possible scheduler-allocator combinations: since we have 5 schedulers and 6 allocators available, we thus performed a total of 90 tests. As we mentioned earlier, each test instance includes 372320 jobs, for a grand total of 33.5 million jobs simulated and analyzed. We will use various metrics

in order to evaluate the results obtained from testing, all of which were introduced in Section 2.1.3.

The Eurora system was modeled and simulated through the AccaSim simulator, presented in Chapter 4. In Eurora's configuration file we defined the two types of nodes in the system, possessing respectively GPU and MIC accelerators, and their amount, which corresponds to 32 nodes for each type. Finally, a parser was implemented in order to correctly read the job entries from the workload, which are written in a custom format designed for the PBS Workload Management System. It should be noted that since Eurora is a system with multiple job queues, and AccaSim currently supports only one queue, we may obtain slightly different results compared to the real system. This, however, will not impact our observations regarding the effectiveness of the dispatching methods being tested.

The tests were performed on a dedicated server machine, equipped with a 16-cores Intel Xeon CPU and 8GB of RAM, and running Linux Ubuntu 16.04. We parallelized them on three different processes, one for each variant of the Eurora workload. Each test instance took on average 5 hours to complete; the only exceptions were the tests performed with the *Constraint Programming-Based* scheduler, which required a time of one to two days.

Additionally, we performed a series of more specific tests: we considered the jobs related to five different months in the Eurora workload, and conducted tests with specific schedulers on them, in order to assess the influence of an allocator on dispatching performance in short time frames.

6.2 Full Workload Tests

In this section we will present the results obtained with the full Eurora workload, and we will analyze them from different points of view.

6.2.1 Results Overview

In Figure 6.1, we can see an overview of the results that were obtained with the different schedulers, in terms of *slowdown*, *queue size*, and *system load ratio*. For fairness, all schedulers were tested with the Simple allocator, as this overview serves the purpose of understanding the macroscopic differences between the scheduling methods.

The box plots depicted here show some key parameters of the metrics' distributions: in particular, the horizontal lines represent the *minimum*, *median* and *maximum* values respectively. The triangles represent instead the *mean* values, and the rectangles show the range between the first and third *quartiles*. We used the logarithmic scale, because of the huge differences between the schedulers' performance. Also, in order to obtain more meaningful results, all jobs which achieved a slowdown of 1 across all test instances were discarded from the distribution.

As it can be seen, the schedulers show enormous differences in terms of performance, up to three orders of magnitude. These differences appear more evident by considering the median and inter-quartile range values: the means, in fact, are more influenced by outlier jobs that achieve bad performance regardless of the scheduling method being used.
In terms of slowdown the clear winner appears to be the CP scheduler, for which most jobs have values lower than 10. Predictably, the worst scheduler is LJF, which is not a good choice for this kind of workload, mostly made of short jobs. Conversely, SJF performs very well, with a performance close to that of the CP scheduler. The EBF and PRB schedulers show instead similar performance, and are in the middle ground.
All schedulers show a distinct improvement when using the real and predicted job durations, thus proving the effectiveness of the latter. PRB is the only scheduler being completely unaffected by the job duration estimation being used: this makes sense, as this method does not consider the job duration in any way, but only its delay compared to the queue's expected waiting time.

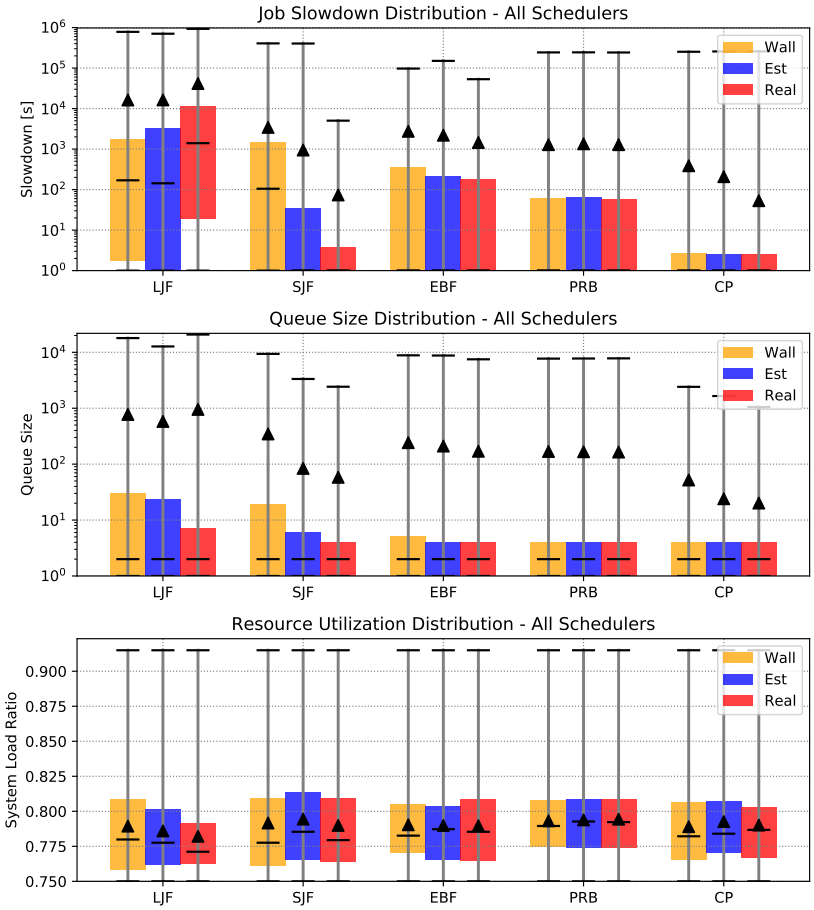


Figure 6.1: Overview of the test results for all schedulers, using the Simple allocator.

In terms of queue size, the differences in performance are more nuanced: while CP is again the best performer, with extremely low median values, most schedulers achieve similar results. The influence of the job duration estimation seems lower as well. Given their similar performance when using the predicted job duration, one may prefer to choose the SJF scheduler over the CP one: while the latter achieves globally optimal results, it is also computationally expensive, while SJF is extremely lightweight.

Lastly, it can be seen that there are very minor differences in terms of system load ratio. In this case, we considered the data related to time steps in which the system had a load ratio higher than 75%, by including all resource

types: this was done to emphasize the ability of each scheduler to keep the system busy. As we can see, however, all schedulers show a very similar behavior, with an average load ratio close to the 78% value. The absence of big differences from this point of view is a curious fact, especially considering that most schedulers show huge differences in terms of throughput.

6.2.2 Slowdown Analysis

We will now analyze more in detail the results we obtained, from the point of view of slowdown. In Figure 6.2 the performance of all schedulers, tested against all allocators, is shown. In this case, since the distributions are quite similar within the same scheduler and the differences in performance are smaller, we will only show the mean values in a linear scale. Like we have done previously, jobs achieving a slowdown of 1 with all allocators for a fixed scheduler were discarded from the computation.

It can be seen that, while the differences are not as big as across schedulers, some allocators may significantly improve the performance of the dispatcher: for example, the PRB and EBF schedulers show an improvement roughly between 15% and 20% when using allocation methods like Balanced and Weighted. These improvements seem more evident when using the wall time and the estimated duration, and they are in general much smaller when using the real one.
LJF has a very peculiar behavior: unlike other schedulers, much better performance is achieved when using the wall time duration, which is counter-intuitive. This happens because LJF highly penalizes short jobs, which are the most common in the Eurora workload, and the most sensible to slowdown as well: by using the wall time duration, these jobs have a higher probability of being picked first from the queue, hence the opposite behavior compared to the other schedulers.

In general, it can be seen that the performance improvement related to the allocators is more evident with low throughput schedulers: LJF has a

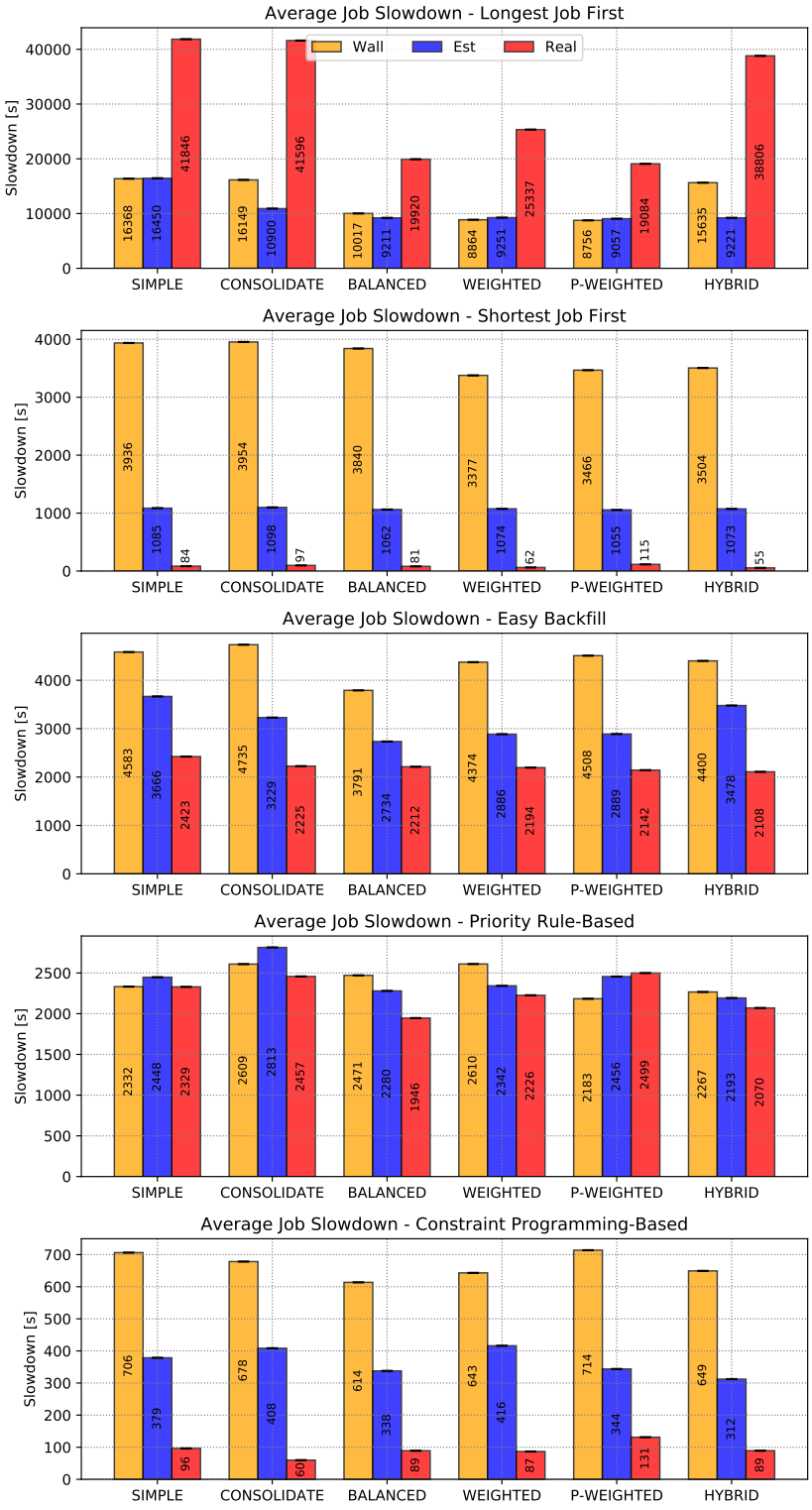


Figure 6.2: Test results in terms of average slowdown for all schedulers.

performance variability of over 50%, while SJF and CP see only minor differences, suggesting us that their behavior is already close to the optimum achievable with this workload. Drawing a winner among the various allocators can be hard, as they show varying behavior: however, Weighted and Priority-Weighted represent the most reliable alternatives, with stable performance across all schedulers. Hybrid seems to be more unpredictable and surprisingly, Consolidate seems to be the worst-performing allocator, often worse even than Simple.

6.2.3 Queue Size Analysis

Queue size is the second parameter we will analyze in order to assess the performance of the various allocation methods. In Figure 6.3 the performance in terms of queue size is shown for each scheduler and allocator: like with the slowdown, we are only showing the mean values, as they are sufficiently descriptive for our analysis.

It seems evident that, for this specific parameter, the allocation method being used has a smaller influence, compared to what we have seen with the slowdown; there are still some differences, especially for the wall time and estimated duration instances, with the biggest variations found again in the LJF scheduler. PRB shows appreciable improvements as well: this last case seems to be rather strange, as the Consolidate allocator shows a distinctly worse average queue size, which is roughly 20% bigger than with the Weighted and Balanced allocators.

The smaller improvements in terms of queue size, compared to what we have seen with the slowdown, are expected. This is in fact a parameter that depends primarily on the scheduler, which has full control over the job queue. The allocator acts instead as a *slave* component with a limited, scheduler-defined view of the queue, and thus having a smaller influence over it.
Still, the fact itself that the allocator policy being used can influence the system's throughput meaningfully is a relevant fact: as we have seen earlier,

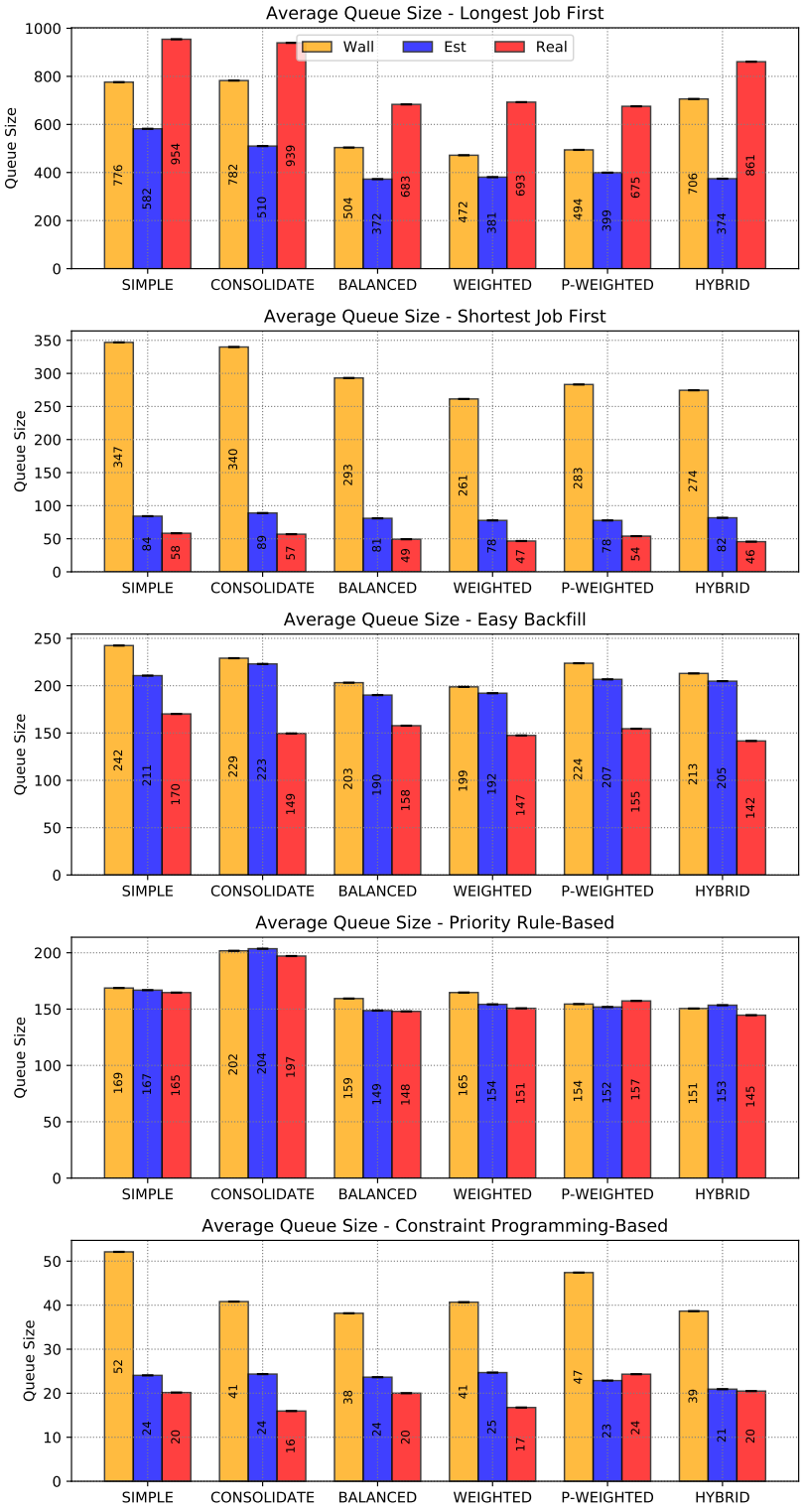


Figure 6.3: Test results in terms of average queue size for all schedulers.

allocation methods are often made with a different purpose in mind, for example to improve job locality or power consumption, while the scheduler alone is usually in charge of maximizing the system's throughput, unlike what happens here.

6.2.4 Resource Allocation Efficiency Analysis

Having characterized the differences in terms of throughput between the various dispatcher configurations, we will now consider how efficiently resources are allocated. In Figure 6.4 the mean resource allocation efficiency values, calculated per-job in each test instance, are presented. Unlike with the slowdown and queue size overview, the differences are not big enough to justify the visualization of all parameters of the distribution, which are very similar across all instances.

As it can be seen, all schedulers behave in a very similar way and show the same kind of trend: this is to be expected, as the resource allocation efficiency depends mostly on the allocation policy being used, rather than the scheduler. There are minor differences though, with LJF predictably scoring lower values than the other schedulers.
With all schedulers, it can be seen that Consolidate is the allocator achieving the highest mean resource allocation efficiency: this makes sense, as Consolidate is a pure best-fit allocator targeted at performing consolidation, while the other methods are more throughput-oriented. Allocators like Priority-Weighted and Hybrid are however able to reach values close to those of Consolidate, which is a good result.

The results shown here are far from optimal: it can be seen that no scheduler-allocator combination is able to reach mean resource allocation efficiency values above 60%, which means that in most cases jobs are allocated with a poor fit, thus leading to high resource fragmentation. This is weird, as scheduling methods like Easy Backfill or the Constraint Programming-Based one are highly performant, and designed to exploit the system's resources

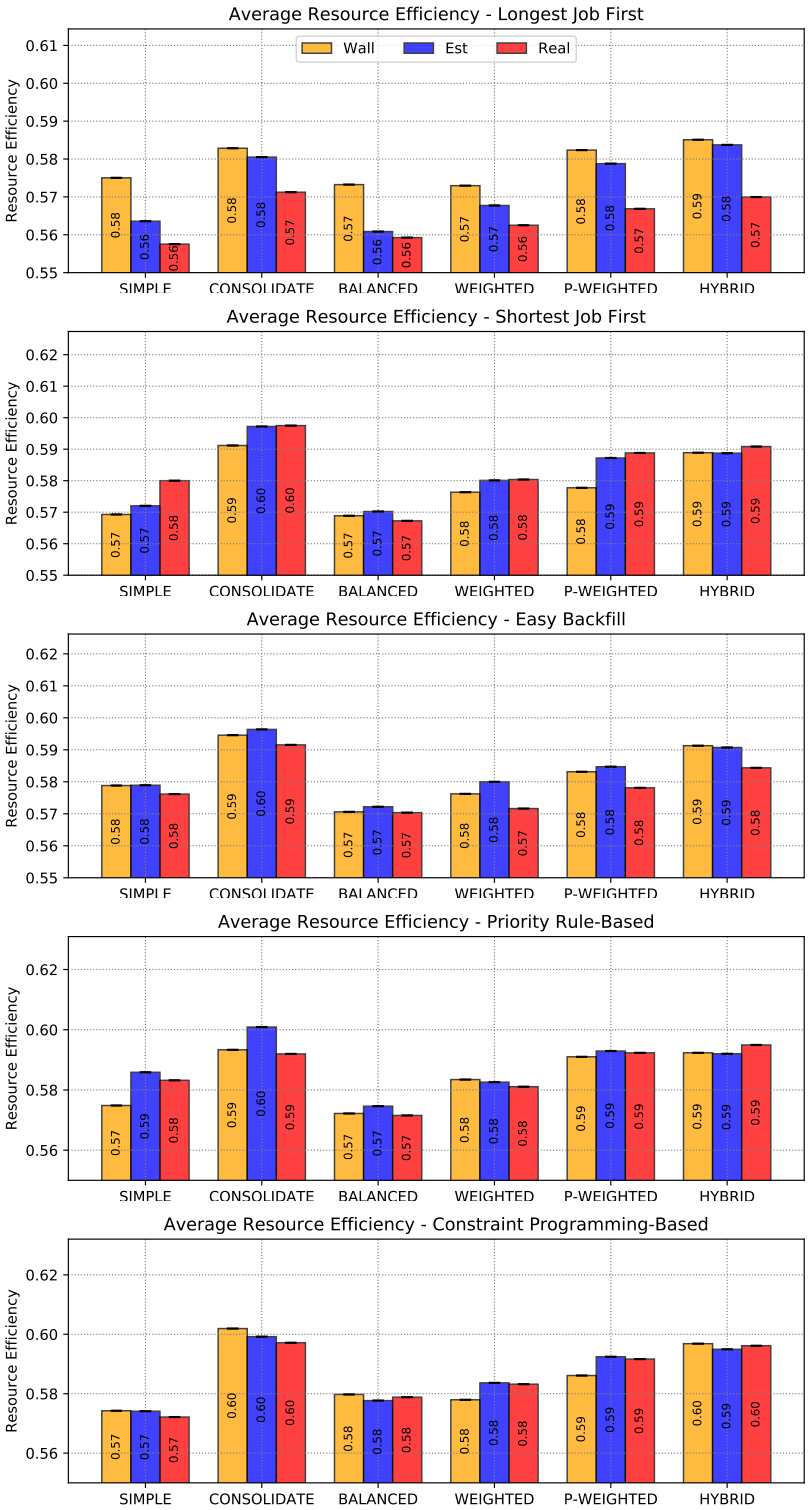


Figure 6.4: Test results in terms of average resource allocation efficiency for all schedulers.

to their maximum at all times. The same goes, again, for allocators like Consolidate. At the same time, the type of job duration estimation being used does not produce significant improvements, with unstable results across the various instances. This leads us to believe that the poor values obtained here do not depend on the dispatching methods themselves, but rather on the system's design and on the workload's structure.

6.2.5 Load Ratio Analysis

We will now perform a more detailed analysis about how the system's resources are used in our workload, and what issues there might be with Eurora's design, preventing the system from reaching optimal resource efficiency values. For our purpose, we will use the per-step variant of the resource allocation efficiency metric, which is similar to the load ratio and able to characterize very well the utilization of resources in the system.
Also, at this stage we will only consider the Constraint Programming-Based scheduler with the Weighted allocator, using the real duration variant of the workload: this was done in order to present an optimal case, as confirmed by the slowdown and queue size results, and to get rid of all dispatching-related influence factors *a priori*. We still compared the results obtained with the other schedulers and allocators, which proved to be similar or slightly worse than the ones shown here.

In Figure 6.5 we present a first plot depicting the resource utilization observed in our workload. Every point is related to a time step in the simulation: the X values represent the number of nodes used by running jobs, while the Y values represent the resource allocation efficiency on such nodes for said jobs. This is a density distribution plot, where darker areas represent values with a higher frequency in the distribution, and vice versa.
This plot is not very informative: we can however see that, almost regardless of the number of used nodes, the resource allocation efficiency is always in the 60%-80% range, sometimes stretching even lower, and thus confirming

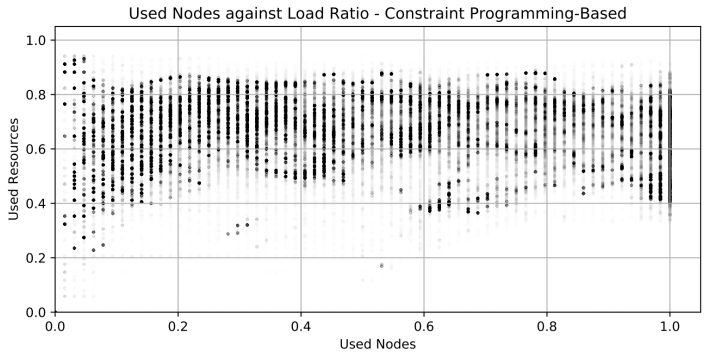


Figure 6.5: Test results in terms of resource efficiency for the CP scheduler with the Weighted allocator, while using the real job duration, for all resource types.

our previous results. The fact that this metric is almost independent from the X values is, again, not a good sign.

Figure 6.6 is much more insightful, as we present the same plot shown earlier, but separated for each resource type in Eurora. It is evident that there is no problem with the GPU and MIC resources: their resource allocation efficiency values are almost always in the 80%-100% range, which is very good and close to optimal. Among the two, MIC resources seem to perform slightly worse, but it is likely due to the extremely low frequency of jobs needing such resources. In fact, there are very few points in which more than 40% of the MIC-equipped nodes are used, which is also coherent with the workload analysis performed in Section 3.2.

There are instead evident issues with the allocation of CPU and memory resources: their resource allocation efficiency values are mostly low, and similar to those of the first plot we have shown. Memory resources, in particular, perform very bad, and their efficiency is mostly in the 40%-80% range. This suggests us that these two resource types are badly used in the system, and that fragmentation is high for them.

We can now draw come conclusions: first of all, there is a glaring issue related to GPU-based jobs. The frequency of such jobs in the workload is too

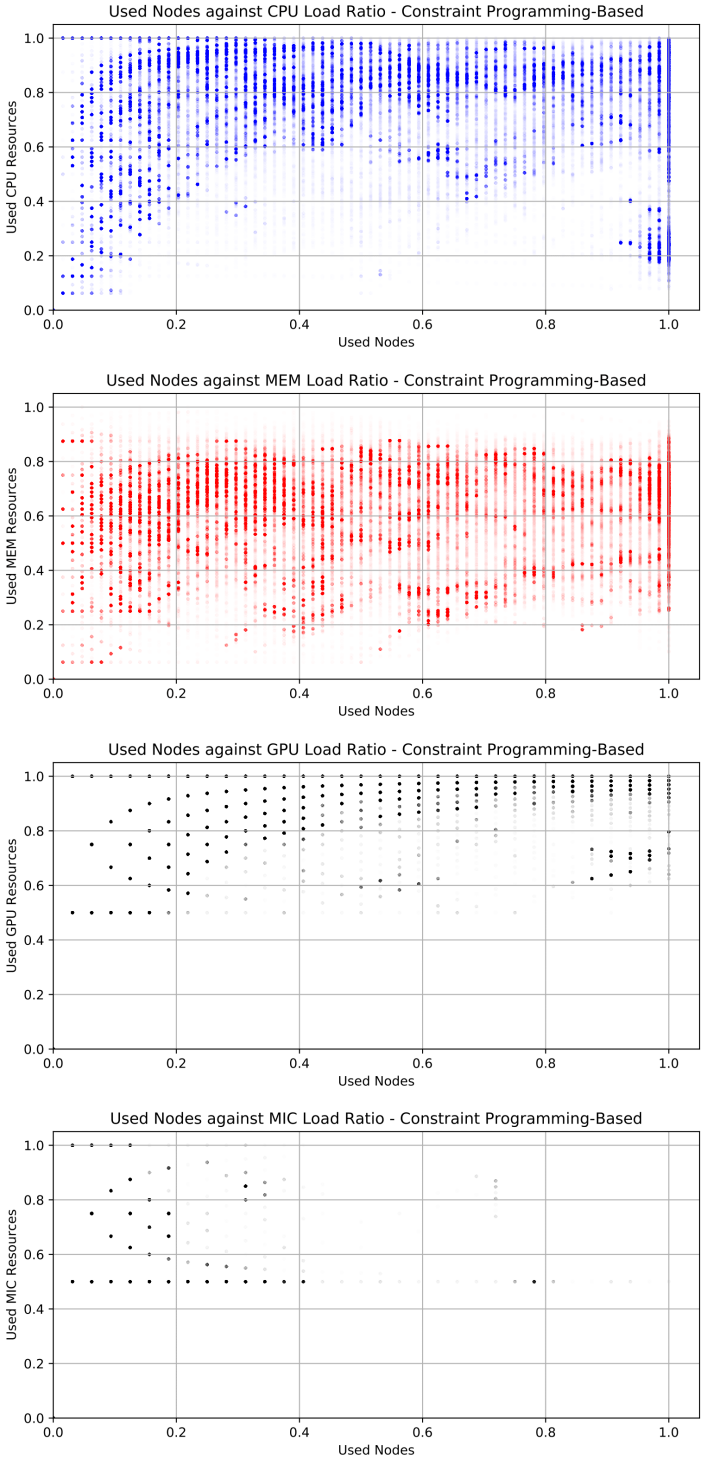


Figure 6.6: Test results in terms of resource efficiency for the CP scheduler with the Weighted allocator, while using the real job duration, divided by resource type.

high compared to the amount of GPU resources available: this means that such resources can act as a bottleneck to the entire system, thus impairing its performance, while MIC resources go unused most of the time. Besides, since there are only 2 GPUs in each node, at most 2 units for a given job can be allocated to a single node: jobs requiring many units will thus be spread over a large quantity of nodes, leaving a great amount of unused resources. The second issue is related to memory management: in the PBS Workload Management System, and in our workload as well, the basic memory allocation unit is 1GB, with each node having 16GB available. This kind of memory allocation policy is too coarse, and will most likely lead to large amounts of wasted memory in the system, as most jobs will be allocated considering a worst-case upper bound. We are confident that a more fine-grained memory allocation policy could dramatically improve the system's throughput.

The low resource allocation efficiency values obtained with the various dispatching configurations in Section 6.2.4, together with their low variability, must then be attributed to a series of system-level issues that can be hardly solved. This is also the reason for which the various schedulers and allocators have such a dramatic effect on the throughput: in a small-scale and delicate system like Eurora, job planning and management of the resources are critical.

6.3 Single Test Cases

Having characterized the system's behavior with the full workload, we will now present some specific test cases, in which certain scheduling methods were used on a subset of the full workload's data. This will allow us to better understand how allocation methods can influence the system's throughput on short time frames.

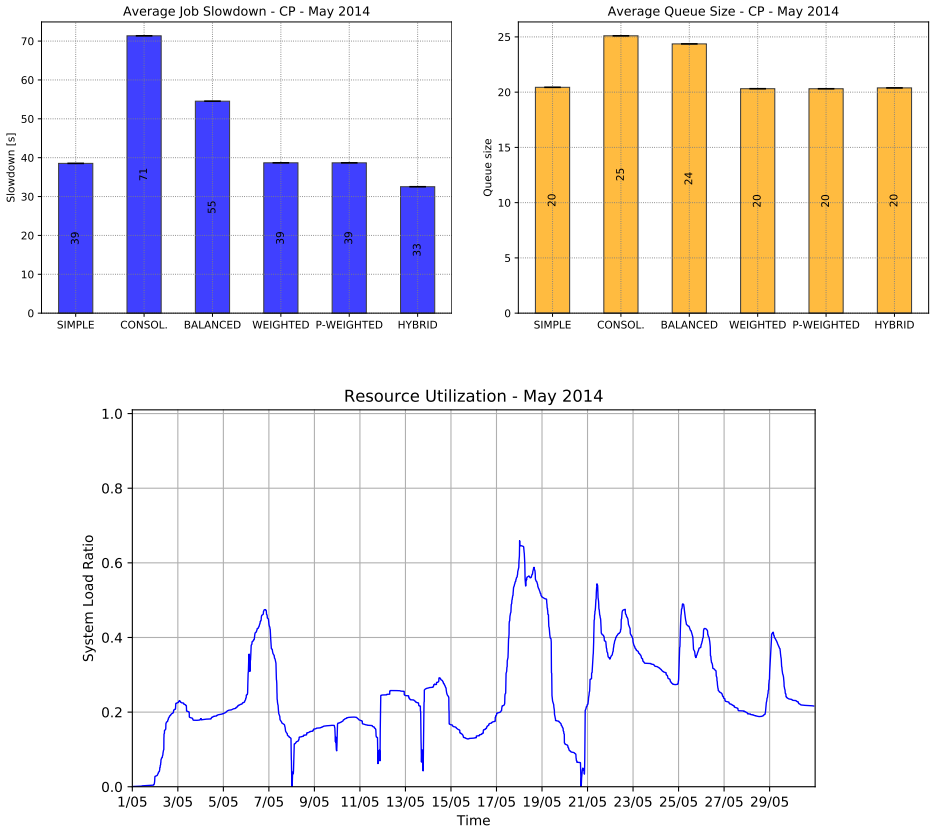


Figure 6.7: Results obtained with the Constraint Programming-Based scheduler, for the May 2014 instance using the wall time duration.

6.3.1 May 2014

Here we consider all jobs which were submitted in May 2014. This workload is made up of 4931 jobs, of which 68.8% are GPU-Based, while 30.8% are of the Standard type, and the remaining 0.4% are MIC-Based. Since the first two classes of jobs are mostly balanced in frequency, this instance is not an excessively hard one. The Constraint Programming-Based scheduler was used, together with the wall time as a job duration estimation.

As shown in Figure 6.7, there are large variations on the results depending on the allocator used. In particular, there is a 50% difference, in terms of slowdown, between the Consolidate and Weighted allocators. Discarding the former, all remaining allocators perform similarly, with Hybrid achiev-

ing optimal performance. Similar improvements can also be seen with the average queue size, even though they are more subtle, as observed earlier.

The resource utilization plot in terms of system load ratio is shown as well for the month considered, and was obtained from the Weighted allocator's instance. It can be seen that the curve itself is rather smooth, since the CP scheduler is able to keep resource utilization high and stable, by planning the schedule in advance. Load ratio values are always low though, which may be related to both the workload being very small, and to the considerations made regarding Eurora's design in Section 6.2.5.

6.3.2 June 2014

The second test instance we are presenting is related to the month of June 2014, and contains 6204 jobs. Of those, roughly 91% are GPU-Based, while 8.6% are Standard, and the remaining 0.4% are MIC-Based. This test was performed with the Shortest Job First scheduler, using the predicted job duration.

The results are shown in Figure 6.8. As it can be seen, the slowdown and queue size average values are low and homogeneous for all allocators, with negligible differences. This is due to both the workload being relatively "easy", with most allocators being able to reach optimum results with the SJF scheduler, and to the astoundingly high percentage of GPU-Based jobs, which leaves little room for improvement on the allocator's side.

The resource utilization plot, obtained again from the Weighted instance, is also quite peculiar: the system load ratio is most of the time at a roughly constant level, with a single, prolonged spike towards the middle of the month. This likely means that the few jobs making up the workload are short and spaced enough between each other, in terms of submission times, to not cause any kind of congestion in the system regardless of the scheduler.

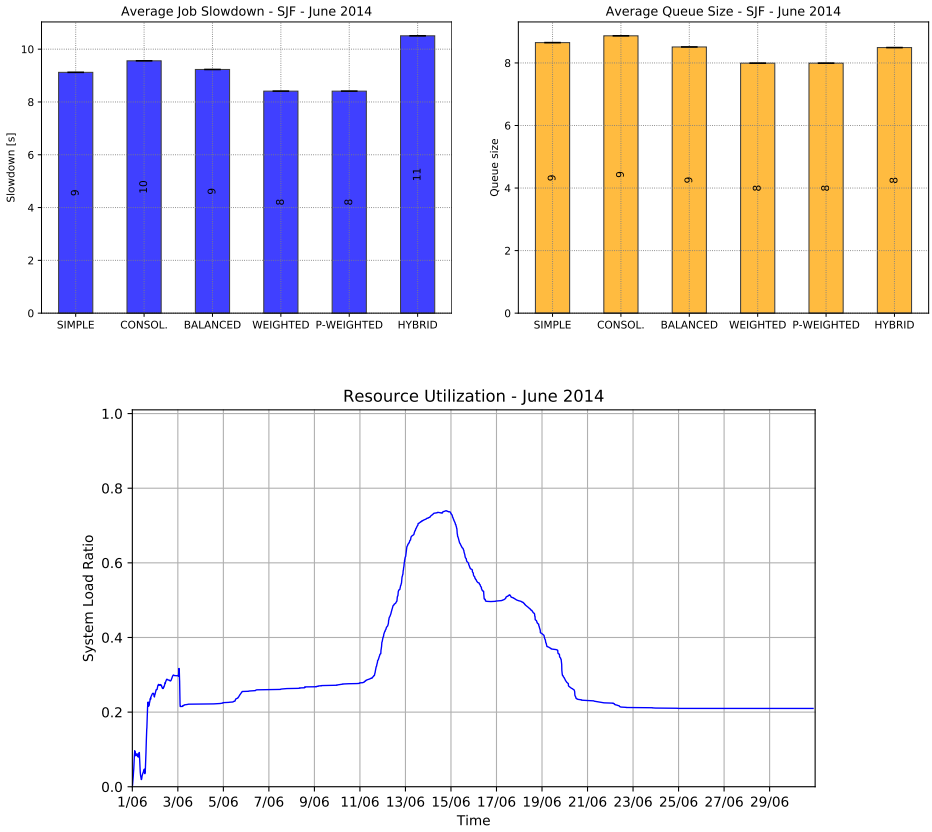


Figure 6.8: Results obtained with the Shortest Job First scheduler, for the June 2014 instance using the predicted job duration.

6.3.3 August 2014

We will consider now the month of August 2014, with 47967 jobs. These are divided in 84.8% GPU-Based jobs, 14.8% Standard jobs, and 0.4% MIC-Based jobs: the workload is similar to the June 2014 one, but much larger in size. This test was performed by using the Easy Backfill scheduler, with the predicted job duration.

In this case, there are again large variations in the results obtained with the various allocators, which are shown in Figure 6.9. While all average slowdown values are quite small, there is a 25% difference between Simple and Weighted, and a 40% one between Consolidate and Weighted. This last allocator is the best-performing one for this month as well. Queue size values

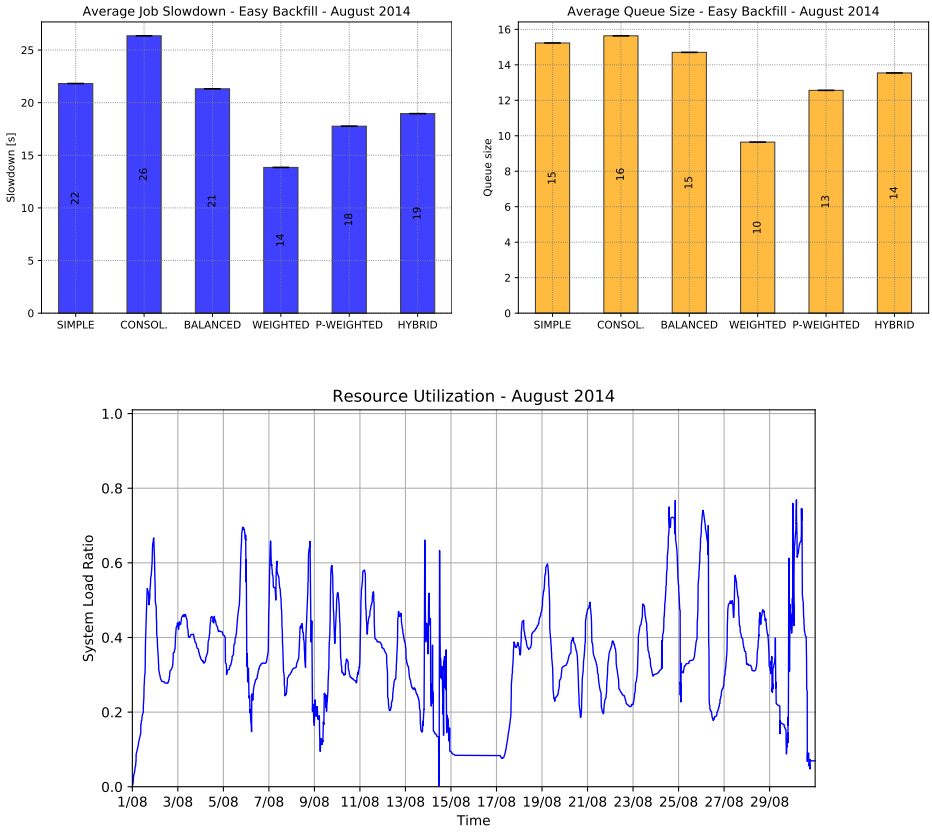


Figure 6.9: Results obtained with the Easy Backfill scheduler, for the August 2014 instance using the predicted job duration.

show a similar trend, confirming Weighted's good performance.

The resource utilization plot, as always related to the Weighted allocator, is much more noisy than the previous ones, with load ratio values almost always lower than the 60% threshold. This is rather surprising as Backfill, like the CP scheduler, is able to efficiently *fill the gaps* in the system's resources to better use them. The spikes are then probably due to the high presence of GPU-Based jobs: even if the system has a low load ratio, GPU-Based jobs cannot be started as long as some accelerator units are not released, thus leading to inefficient resource utilization.

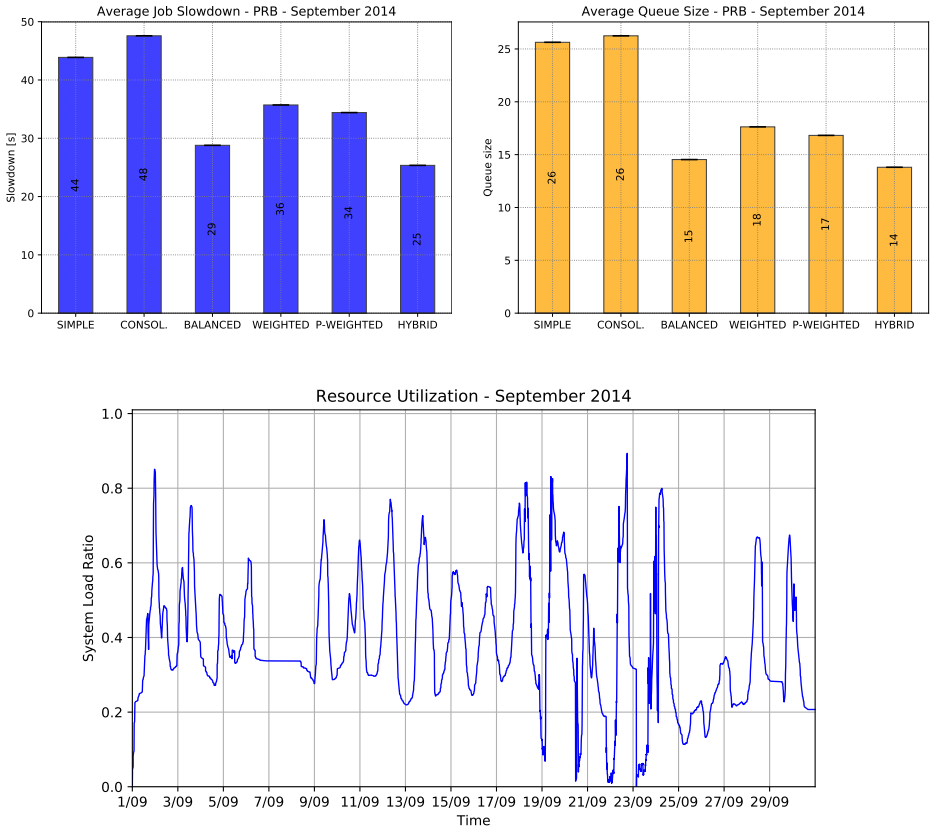


Figure 6.10: Results obtained with the Priority Rule-Based scheduler, for the September 2014 instance using the wall time duration.

6.3.4 September 2014

The test instance analyzed in this section is related to the September 2014 month, and contains 77786 jobs. Of those, 87.8% are again GPU-Based, 11.4% are of the Standard type, and 0.8% are MIC-Based. This workload is very similar to the August 2014 one: in order to obtain meaningfully different results, we experimented with the Priority Rule-Based scheduler, using the wall time duration.

The results depicted in Figure 6.10 show again meaningful differences between the various allocators. Balanced, together with its refinement Hybrid, are the best-performing allocators, with 40% lower average slowdown values compared to Simple and Consolidate. The same differences can be seen in

the average queue sizes, and are more pronounced than in the other tests. Weighted and Priority-Weighted do slightly worse than Balanced, but are still better than Simple and Consolidate.

The resource utilization plot, obtained with the Weighted allocator, is similar to the one seen for the August 2014 instance, thus confirming the similarity between the two workloads, even though the schedulers being used are different. In this case, much higher load ratio values are reached, going up to 80%: this is likely due to the comparatively higher presence of jobs based on CPU and memory alone, which can be allocated to the nodes equipped with MIC units.

6.3.5 January 2015

The last workload we will analyze considers the January 2015 month, and is made of 46768 jobs: of these, 93.4% are GPU-Based, 6.4% are Standard, and only 0.2% are MIC-Based. The Priority Rule-Based scheduler was again used, this time with the predicted job duration.

The results in Figure 6.11 are quite extreme: most allocators achieve average slowdown values 50% lower than the Simple allocator, and almost an order of magnitude lower than Consolidate. The queue size results reflect those seen with the slowdown, with large differences being observed.
These results should not be taken as an example of the influence an allocation method can make on system throughput, as they are exceptional: however, they prove that allocation heuristics not designed with the target system in mind can lead to disastrous results. This applies in particular to Consolidate, which has very bad performance in this case, despite being a theoretically optimal best-fit allocator.

The plot for resource utilization is quite peculiar, and was obtained again with the Weighted allocator. The curve is mostly smooth, and shows a rising trend from the start to the end of the month. This is likely an indicator of the presence of very long jobs, both Standard and GPU-Based: in this case,

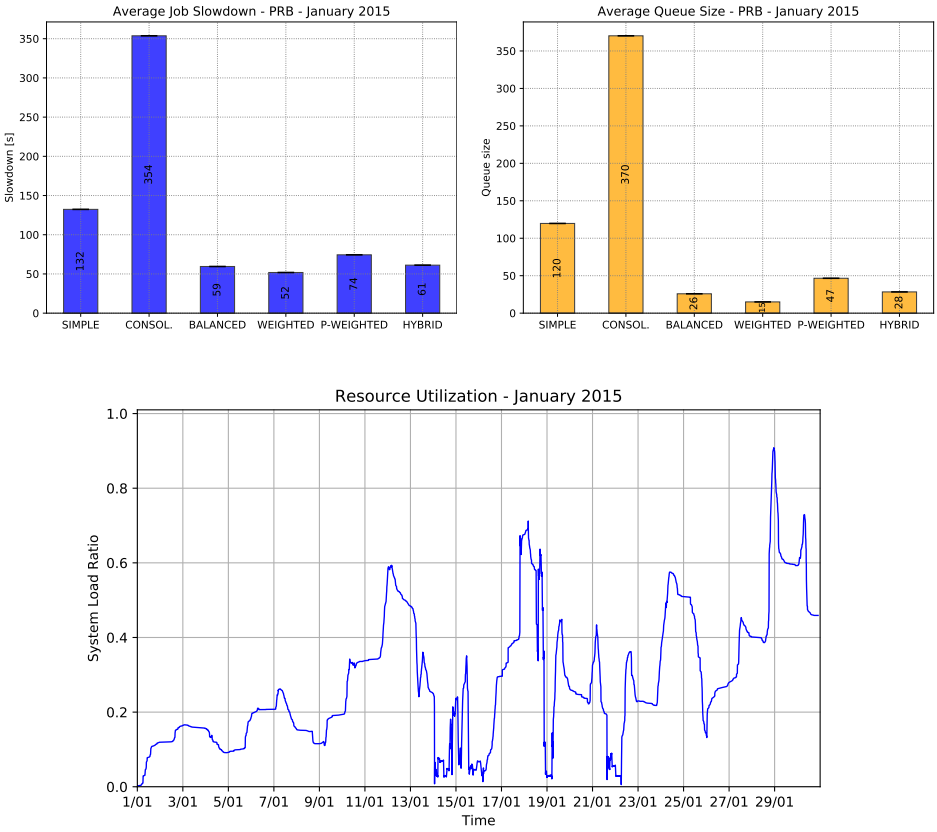


Figure 6.11: Results obtained with the Priority Rule-Based scheduler, for the January 2015 instance using the predicted job duration.

managing the resources efficiently is vital, as poor allocations can lead to large numbers of jobs piling up in the queue, and to the bad results shown here for the Simple and Consolidate allocators.

6.4 Experimental Observations

The experiments we performed show that the adoption of certain allocation heuristics can substantially improve the **throughput** of an HPC system, and thus the *Quality of Service* offered by it. Our claims are related to heterogeneous and small-scale systems in particular, in which the management of certain resource types is vital, but can be extended to homogeneous or large-scale systems as well.

We have also shown that the impact of an allocator on the throughput is more limited on **long time frames**, in the order of years: the overall, macroscopic behavior of the system is in this case dictated by the scheduler, with light QoS improvements given by the allocator which can be up to 20%. At the same time, we noticed bigger improvements with low-performance schedulers, while smaller variations have been observed with high-performance scheduling algorithms. This applies to the kind of job duration estimation being used as well, with greater room for improvement observed when using low-accuracy measures like the jobs' wall time. When considering **short time frames** like months, however, the allocator has a much bigger importance, and can dramatically influence the throughput of the system, thus impacting directly the QoS perceived by users. We even observed differences of one order of magnitude in throughput, which are very significant.
All of these results depend, of course, on the workload's typology: cases of particularly difficult instances that result in bad QoS regardless of the scheduler and allocator being used are not rare, but they should not be taken much into account, because of their exceptional nature.

We have also observed and explained the **flaws** present in Eurora's design and in its workload, which limit the system's performance and efficiency: we are referring, in particular, to the severe imbalance between the availability of GPU resources, and the frequency of GPU-Based jobs in the workload. The system was, in fact, designed under the assumption of uniform usage of GPU and MIC resources. In this small-scale, delicate, resource-critical environment the allocator's role gains great importance, and it has proven to be a fundamental component in the optimization of a Workload Management System.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

In this work we have designed, implemented and analyzed various dispatching methods targeted at heterogeneous HPC systems which, as we have seen, are both more powerful and more delicate than their homogeneous counterparts, and require better care in order for their potential to be fully exploited; our analysis considered the **Eurora** HPC system, and was focused on throughput evaluation specifically. We dedicated particular attention to **allocation** heuristics, as most of the scheduling algorithms being used were already available and just needed minor tweaking and improvement work. We have also used and analyzed various methods for job duration estimation, with particular attention on a novel data-driven prediction heuristic, which has proven to be highly beneficial to system throughput.
Besides, we have contributed to the development of several core components of **AccaSim**, an HPC Workload Management System simulator which has been developed in the University of Bologna, and used in the context of our research. Lastly, as a natural step, we have analyzed the Eurora system and its workload: this analysis allowed us to show its peculiarities and weak points, besides making the behavior of our algorithms more comprehensible and scientifically explainable. This will allow us, in turn, to make our

considerations independent from the specific system being used and its scale.

Our work has shown the importance of allocation heuristics in HPC systems: in a heterogeneous, small-scale context, an allocation heuristic designed with a particular system in mind can significantly improve the Quality of Service offered by it, and a good scheduler alone is not enough to achieve this goal. We have also seen that a good heuristic for the prediction of the jobs' duration can improve the throughput by several orders of magnitude, even though this is closely related to the scheduling policy being used, rather than the allocation one.
We have, specifically, observed a certain duality while analyzing the effect of allocation heuristics on throughput: while on long time frames, like years, only small variations in throughput can be observed, up to 20%, much bigger ones can be seen when considering *short* time frames, in the order of months, which also correspond to the users' perception of the system. We even observed throughput variations as large as one order of magnitude: these lead to a fundamental change in user experience with the HPC system being considered, due to its online and interactive nature, and are therefore very significant. This duality between short and long time frames also suggests us that the system's behavior is in the long term dominated by the scheduling method being used, with the allocator acting as a slave component, in accordance to the architecture model we proposed.

Our analysis of the Eurora system also allowed us to understand how delicate heterogeneous HPC systems are, and how easy it is for them to be used in inefficient ways, which lead to resource under-utilization, high waiting times, and low energy efficiency. This stems, in our specific case, from the interaction of the system with a workload with certain statistical features, such as the very high frequency of GPU-Based jobs in Eurora's, which go against its design assumptions. Thanks to this analysis, we are again able to fully comprehend the behavior of our algorithms, and thus extend our considerations to larger-scale, different contexts as well, and to a great variety of systems.

7.2 Future Work

Future work will be targeted at testing the implemented allocation heuristics' effectiveness with different systems, and with different workloads, as we need to get past the limits that lie within Eurora. This system was, in fact, thoroughly explored and analyzed, and there is a need to move towards a larger-scale context. We will also experiment with new, different allocation heuristics more akin to the *mapping* type, in order to see how they compare against the ones we developed, which are mainly designed to be lightweight. Lastly, while already fast and efficient, our heuristics could see some optimization work, in order to further improve their scalability.
As for AccaSim, the future plans for development are aimed at improving existing features and adding new ones, as the core of the simulator is a solid base. First of all, we plan to improve the visualization tools, by adding features aimed at producing specific plots in order to evaluate the results. We also plan to improve the resource usage monitoring features and, most importantly, a synthetic job generation mechanism will be added, in order to grant more flexibility to the simulator. This will allow us, in fact, to create workloads with specific statistical features, useful for our research, without needing actual data which may be difficult to obtain. Lastly, we plan to add native support for systems with multiple job queues.

Our work regarding the AccaSim simulator was presented at the *Latin America High-Performance Computing Conference* (CARLA 2017), held on September 2017 in Buenos Aires, Argentina, with the paper *"AccaSim: an HPC Simulator for Workload Management"*. A second paper, focused on our experimental results, is currently in the works and expected to be submitted for the *15th International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research* (CPAIOR 2018), which will be held on June 2018 in Delft, Netherlands.

Bibliography

[3] I. Foster, Y. Zhao, I. Raicu, and S. Lu. "Cloud computing and grid computing 360-degree compared". In: *Grid Computing Environments Workshop, 2008. GCE'08*. Ieee. 2008, pp. 1–10.

[4] H. Fu, J. Liao, J. Yang, L. Wang, Z. Song, X. Huang, C. Yang, W. Xue, F. Liu, F. Qiao, et al. "The Sunway TaihuLight supercomputer: system and applications". In: *Science China Information Sciences* 59.7 (2016), pp. 1–16.

[5] A. Bartolini, M. Cacciari, C. Cavazzoni, G. Tecchiolli, and L. Benini. "Unveiling Eurora—Thermal and power characterization of the most energy-efficient supercomputer in the world". In: *Design, Automation and Test in Europe Conference and Exhibition (DATE), 2014*. IEEE. 2014, pp. 1–6.

[6] C. Galleguillos, Z. Kiziltan, and A. Netti. "AccaSim: an HPC Simulator for Workload Management". In: *To appear in High-Performance Applications and Tools in Latin America High-Performance Computing Conference*. 2017.

[7] A. Borghesi. "Power-Aware Job Dispatching in High-Performance Computing Systems". In: *Tesi di Dottorato di Ricerca in Computer Science and Engineering, Università degli Studi di Bologna* (2017).

[8] Dror G Feitelson. "Metrics for parallel job scheduling and their convergence". In: *Lecture Notes in Computer Science* 2221 (2001), pp. 188–206.

[9] C. Lu, J. Browne, R. DeLeon, J. Hammond, W. Barth, T. Furlani, S. Gallo, M. Jones, and A. Patra. "Comprehensive job level resource usage measurement and analysis for XSEDE HPC systems". In: *Proceedings of the Conference on Extreme Science and Engineering Discovery Environment: Gateway to Discovery*. ACM. 2013, p. 50.

[10] M. Hovestadt, O. Kao, A. Keller, and A. Streit. "Scheduling in HPC resource management systems: Queuing vs. planning". In: *Job Scheduling Strategies for Parallel Processing*. Springer. 2003, pp. 1–20.

[11] J. Du and J. Leung. "Minimizing total tardiness on one machine is NP-hard". In: *Mathematics of operations research* 15.3 (1990), pp. 483–495.

[12] R. Haupt. "A survey of priority rule-based scheduling". In: *OR spectrum* 11.1 (1989), pp. 3–16.

[13] A. Wong and A.M. Goscinski. "Evaluating the easy-backfill job scheduling of static workloads on clusters". In: *Cluster Computing, 2007 IEEE International Conference on*. IEEE. 2007, pp. 64–73.

[14] Q. Snell, M. Clement, and D. Jackson. "Preemption based backfill". In: *Job Scheduling Strategies for Parallel Processing*. Springer. 2002, pp. 24–37.

[15] T. Braun, H.J. Siegel, N. Beck, L. Booni, M. Maheswaran, A. Reuther, J. Robertson, M. Theys, B. Yao, D. Hensgen, et al. "A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems". In: *Journal of Parallel and Distributed computing* 61.6 (2001), pp. 810–837.

[16] J. Meng, S. McCauley, F. Kaplan, V. Leung, and A. Coskun. "Simulation and optimization of HPC job allocation for jointly reducing communication and cooling costs". In: *Sustainable Computing: Informatics and Systems* 6 (2015), pp. 48–57.

[17] C. Bash and G. Forman. "Cool Job Allocation: Measuring the Power Savings of Placing Jobs at Cooling-Efficient Locations in the Data Center." In: *USENIX Annual Technical Conference*. Vol. 138. 2007, p. 140.

[18] A. Verma, P. Ahuja, and A. Neogi. "Power-aware dynamic placement of hpc applications". In: *Proceedings of the 22nd annual international conference on Supercomputing*. ACM. 2008, pp. 175–184.

[19] J. Hursey, J. Squyres, and T. Dontje. "Locality-aware parallel process mapping for multi-core HPC systems". In: *Cluster Computing (CLUSTER), 2011 IEEE International Conference on*. IEEE. 2011, pp. 527–531.

[25] G.F. Pfister. "An introduction to the infiniband architecture". In: *High-Performance Mass Storage and Parallel I/O* 42 (2001), pp. 617–632.

[26] C. Galleguillos, A. Sirbu, Z. Kiziltan, O. Babaoglu, A. Borghesi, and T. Bridi. "Data-driven job dispatching in HPC systems". In: *To appear in Proceedings of the Third International Conference on Machine Learning, Optimization and Big Data (MOD 2017)*. 2017.

[27] M. Mubarak, C. Carothers, R. Ross, and P. Carns. "Enabling parallel simulation of large-scale hpc network systems". In: *IEEE Transactions on Parallel and Distributed Systems* 28.1 (2017), pp. 87–100.

[28] C. Gomez-Martin, M. Vega-Rodriguez, and J.L. Gonzalez-Sanchez. "Performance and energy aware scheduling simulator for HPC: evaluating different resource selection methods". In: *Concurrency and Computation: Practice and Experience* 27.17 (2015), pp. 5436–5459.

[29] G. Rodrigo, E. Elmroth, P.O. Ostberg, and L. Ramakrishnan. "ScSF: A Scheduling Simulation Framework". In: *Workshop on Job Scheduling Strategies for Parallel Processing. Accepted, Springer*. 2017.

[30] J. Brennan, I. Kureshi, and V. Holmes. "CDES: an approach to HPC workload modelling". In: *Distributed Simulation and Real Time Applications (DS-RT), 2014 IEEE/ACM 18th International Symposium on*. IEEE. 2014, pp. 47–54.

[31] W.B. Hurst. *Modeling and simulation of hpc systems through job scheduling analysis*. University of Arkansas at Little Rock, 2011.

Sitography

[1] *HPC Advisory Council - An Introduction to HPC*. URL: http://www.hpcadvisorycouncil.com/pdf/Intro_to_HPC.pdf.

[2] *Top500*. URL: https://www.top500.org/.

[20] *SLURM - A highly scalable workload manager*. URL: https://github.com/SchedMD/slurm.

[21] *PBS Professional*. URL: http://www.pbsworks.com/PBSProduct.aspx?n=PBS-Professional&c=Overview-and-Capabilities.

[22] *TORQUE Resource Manager*. URL: http://www.adaptivecomputing.com/products/open-source/torque/.

[23] *MAUI Cluster Scheduler*. URL: http://www.adaptivecomputing.com/products/open-source/maui/.

[24] *Eurora - CINECA*. URL: https://www.cineca.it/it/content/eurora.

[32] *Omnet++ - Discrete Event Simulator*. URL: https://omnetpp.org/.

[33] *AccaSim on GitHub*. URL: https://github.com/cgalleguillosm/accasim.

[34] *Standard Workload Format*. URL: http://www.cs.huji.ac.il/labs/parallel/workload/swf.html.

[35] *HPC2N - Seth*. URL: https://www.hpc2n.umu.se/resources/hardware/seth.

[36] *The HPC2N Seth log*. URL: http://www.cs.huji.ac.il/labs/parallel/workload/l_hpc2n/index.html.

[37] *Google Optimization Tools*. URL: https://developers.google.com/optimization/.

[38] *TimSort - Wikipedia*. URL: https://en.wikipedia.org/wiki/Timsort.

Acknowledgements

Now that we have reached the end of the thesis, I would like to thank a few people that have helped me, directly or indirectly, in this journey.
First of all, I would like to thank my relator, Prof. Ozalp Babaoglu, who has patiently followed me and given me this opportunity, and to whom I will always look at when working to become a man of science. I would also like to thank everyone at the Department of Computer Science and Engineering of the University of Bologna, specifically Prof. Zeynep Kiziltan, who has been my co-relator, Cristian Galleguillos, and Alina Sîrbu. Working with you has been a pleasant and exciting experience, and has taught me priceless knowledge about research, which I am surely going to need in the future. I also would like to give my thanks to Andrea Borghesi and Prof. Michela Milano, for their work on the Eurora system.

I would like to thank all of the friends that I've made here in Bologna, with a special mention for the Mascarella guys, thanks to whom these two years were among the funniest in my life. I have learned and experienced so much in so little time, and I am not the same person that first came to live here two years ago anymore. I would then like to thank my family, which allowed me to study away and supported me in this experience far from home. A very special thanks goes to Julia, for a number of reasons way too big to fit in a single page, but which I'm sure she will know very well. Our journey has just only barely started.

At last, thanks to everybody I may have forgotten to mention and thanks to you, who are reading this right now.

Ringraziamenti

Ora che abbiamo raggiunto la fine della tesi, vorrei ringraziare alcune delle persone che mi hanno aiutato in questo lungo percorso, in modo diretto o indiretto.
Prima di tutto, vorrei ringraziare il mio relatore, il Prof. Ozalp Babaoglu, il quale mi ha offerto quest'opportunità e pazientemente seguito nel lavoro di tesi, ed il quale sarà sempre il mio punto di riferimento e modello quando mi impegnerò per diventare un uomo di scienza. Vorrei inoltre ringraziare tutti i ricercatori al Dipartimento di Informatica - Scienza e Ingegneria dell'Università di Bologna, ed in particolare la Prof.ssa Zeynep Kiziltan, la quale ha avuto il ruolo di co-relatore per la mia tesi, Cristian Galleguillos, ed Alina Sîrbu. Lavorare con voi è stata un'esperienza piacevole, serena e stimolante, e sono sicuro che la conoscenza che ho acquisito nell'ambito della ricerca si rivelerà inestimabile per me in futuro. I miei ringraziamenti vanno poi ad Andrea Borghesi ed alla Prof.ssa Michela Milano, per il lavoro svolto con il sistema Eurora.

Vorrei ringraziare tutti i miei amici qui a Bologna, con una menzione speciale per i ragazzi di Mascarella, grazie ai quali questi due anni sono stati tra i più divertenti della mia vita. Ho imparato e vissuto così tanto in così poco tempo e, certamente, non sono più la stessa persona che due anni fa è arrivata in questa città. Vorrei poi ringraziare la mia famiglia, la quale mi ha permesso di studiare altrove, e mi ha supportato in questa esperienza lontano da casa. Un ringraziamento speciale va a Julia, per una quantità di ragioni troppo grande per poter entrare in una singola pagina, ma le quali sono sicuro lei conoscerà molto bene. Il nostro viaggio è soltanto appena iniziato.

Infine, vorrei ringraziare chiunque potrei aver dimenticato di nominare, e vorrei ringraziare te, che stai leggendo proprio in questo momento.