Alma Mater Studiorum – Università di Bologna
in cotutela con
Pontificia Universidad Católica de Valparaíso

DOTTORATO DI RICERCA IN
Computer Science and Engineering

Ciclo XXXII

Settore Concorsuale: 01/B1
Settore Scientifico Disciplinare: INF/01

CONSTRAINT PROGRAMMING-BASED JOB DISPATCHING FOR MODERN HPC APPLICATIONS

Presentata da: Cristian Alejandro Galleguillos Miccono

Coordinatore Dottorato
Prof. Davide Sangiorgi

Supervisore
Prof.ssa Zeynep Kiziltan

Supervisore
Prof. Ricardo Soto

Esame finale anno 2020

Abstract

A High-Performance Computing (HPC) job dispatcher is a critical software that assigns the finite computing resources of a system to the jobs submitted by users, who request some of such computing resources to execute their software. This assignment over time is known as the on-line job dispatching problem in HPC systems. The fact the problem is on-line means that solutions, generated by a job dispatcher, must be computed in real-time, and the required time to generate them cannot exceed some threshold to do not affect the normal system functioning. In addition, the job dispatcher must deal with a lot of uncertainty: unknown submission times, an unknown quantity of requested resources, and unknown (actual) duration of jobs. Heuristic techniques have been broadly used in HPC systems, at the cost of achieving (sub-)optimal solutions in a short time. These heuristics are composed of two separate elements, the scheduling part, and the resource allocation part, thus generate a decoupled decision being the major culprit of performance loss. Optimization techniques are less used for this problem, although they can significantly improve the performance of HPC systems at the expense of higher computation time.

Nowadays, HPC systems are being used for modern applications, such as big data analytics and predictive model building, that employ, in general, many short jobs. Usually, HPC users tend to overestimate the duration of jobs, making it challenging to identify short jobs at dispatching time. However, prediction methods may be useful to improve the accuracy of the expected duration and classify jobs correctly. Therefore, HPC job dispatchers need to process large numbers of short jobs quickly and make decisions on-line while ensuring high Quality-of-Service (QoS) levels and meet demanding response times to generate dispatching decisions. Constraint Programming (CP) has been shown to be an effective approach to tackle job dispatching problems. However, state-of-the-art CP-based job dispatchers are unable to satisfy the challenges of on-line dispatching, such as generate dispatching decisions in a brief period and integrate current and past information of the housing system. Both limitations jeopardize achieving high QoS levels and thus impede the adoption of CP-based dispatchers in HPC systems.

Given the previous reasons, the purpose of this work is to propose a class of CP-based dispatchers that are more suitable for HPC systems running modern applications. To identify the class of jobs, we propose a method to predict job durations, allowing to include more online information in dispatchers. The job dispatchers we propose are able to reduce the time required for generating online dispatching decisions significantly, and are able to make effective use of job duration predictions to decrease waiting times and job slowdowns, especially for workloads dominated by short jobs.

Resumen

Un *job dispatcher* de un sistema computación de alto rendimiento – High Performance Computing (HPC) es un software crítico que asigna los recursos computacionales finitos del sistema a los *jobs* enviados por sus usuarios, quienes solicitan algunos de dichos recursos para ejecutar sus programas. Esta asignación a lo largo del tiempo se conoce como el *on-line job dispatching problem* en sistemas HPC. El hecho de que el problema sea *on-line* significa que las soluciones, generadas por un *job dispatcher*, deben calcularse en tiempo real, y el tiempo requerido para generarlas no puede exceder algún umbral para no afectar el funcionamiento normal del sistema. Además, el *job dispatcher* debe lidiar con mucha incertidumbre: tiempos de envío desconocidos, una cantidad desconocida de recursos solicitados y una duración desconocida (real) de los trabajos. Las técnicas heurísticas se han utilizado ampliamente en los sistemas HPC, a costa de lograr soluciones (sub) óptimas en poco tiempo. Estas heurísticas se componen de dos elementos separados, la parte de *scheduling* y la parte de *resource allocation*, por lo que generan una decisión desacoplada que es el principal culpable de la pérdida de rendimiento. Las técnicas de optimización se utilizan menos para este problema, aunque pueden mejorar significativamente el rendimiento de los sistemas HPC a expensas de un mayor tiempo de cálculo.

Actualmente, los sistemas HPC se utilizan cada vez más para aplicaciones modernas, como el análisis de *big data* y la construcción de modelos predictivos, los cuales emplean generalmente muchos *jobs* de corta duración. En general, los usuarios de HPC tienden a sobreestimar la duración de los *jobs*, por lo que es difícil identificar *jobs* de corta duración al momento del despacho. Sin embargo, los métodos de predicción pueden ser útiles para mejorar la precisión en la duración esperada y clasificar los *jobs* correctamente. En estos escenarios de aplicación, los *job dispatchers* de HPC necesitan procesar grandes cantidades de *jobs* de corta duración rápidamente y tomar decisiones *on-line* al mismo tiempo que garantizan altos niveles de calidad de servicio y cumplir con los exigentes tiempos de respuesta para generar decisiones de despacho. La Programación con Restricciones – Constraint Programming (CP) ha demostrado ser un enfoque eficaz para abordar los problemas de *job dispatching.* Sin embargo, los *job dispatchers* de última generación basados en CP no pueden satisfacer los desafíos del despacho *on-line*, como generar decisiones de despacho en poco tiempo e integrar información actual y pasada del sistema. Ambas limitaciones ponen en peligro el logro de altos niveles de QoS y, en consecuencia, impiden la adopción

de *job dispatchers* basados en CP en los sistemas HPC.

Por estas razones, el propósito de este trabajo es proponer una clase de *job dispatchers* basados en CP que son más adecuados para sistemas HPC que ejecutan aplicaciones modernas. Para identificar la clase de trabajos, proponemos un método para predecir la duración de los *jobs*, lo que permite incluir más información *on-line* en los *job dispatchers*. Los *job dispatchers* que proponemos pueden reducir el tiempo requerido para generar decisiones de despacho *on-line* de manera significativa, y pueden hacer un uso efectivo de las predicciones de duración de los *jobs* para disminuir los tiempos de espera y sus *slowdown*, especialmente para *workloads* dominadas por *jobs* de corta duración.

Riassunto

Un *job dispatcher* di un sistema di calcolo ad elevate prestazioni – High Performance Computing (HPC) è un software critico che alloca le risorse di calcolo finite del sistema ai *jobs* inviati dagli utenti, che richiedono alcune di questi risorse per eseguire il proprio software. Questa assegnazione nel tempo è nota come *on-line job dispatching problem* sui sistemi HPC. Il fatto che il problema sia on-line significa che: le soluzioni, generate da un *job dispatcher*, devono essere calcolate in tempo reale e il tempo necessario per generarle non può superare una determinata soglia in modo da non influire sul normale funzionamento del sistema. Inoltre, un *job dispatcher* deve affrontare molte incertezze, quali: tempi di spedizione sconosciuti, una quantità sconosciuta di risorse richieste e una durata (effettiva) sconosciuta dei lavori. Le tecniche euristiche sono state ampiamente utilizzate nei sistemi HPC, al costo di raggiungere soluzioni (sub-) ottimali in breve tempo. Queste tecniche sono composte da due elementi separati, la parte di *scheduling* e la parte di *resource allocation*, fattori che possono generare una decisione scollegata che è la principale causa della perdita di prestazioni. Le tecniche di ottimizzazione, sebbene possano migliorare significativamente le prestazioni dei sistemi HPC a spese di un tempo di calcolo più lungo, sono meno utilizzate per questo problema.

Attualmente, i sistemi HPC sono sempre più utilizzati per applicazioni moderne, come l'analisi di *big data* e per la costruzione di modelli predittivi, che impiegano di solito molti *jobs* a breve termine. In generale, gli utenti HPC tendono a sovrastimare la durata di *jobs*, quindi è difficile identificare *jobs* di breve durata al momento della spedizione. Tuttavia, i metodi di previsione possono essere utili per migliorare l'accuratezza nella durata prevista e classificare correttamente i *jobs*. In questi scenari applicativi, un HPC *job dispatcher* deve spedire rapidamente grandi quantità di *jobs* a breve termine e prendere decisioni *on-line* garantendo alti livelli di qualità del servizio (QoS). Inoltre, dovrà soddisfare i tempi di risposta impegnativi per generare decisioni di spedizione. La Programmazione a vincoli, Constraint Programming (CP), ha dimostrato di essere efficace per affrontare i problemi di job dispatching. Tuttavia, i *job dispatchers* di ultima generazione basati su CP, non possono far fronte alle sfide di invio *on-line*, come la generazione di decisioni di invio in breve tempo e l'integrazione delle informazioni di sistema attuali e passate. Entrambe le limitazioni potrebbero ostacolare il raggiungimento di alti livelli di QoS e, di conseguenza, impedire l'adozione di *job dispatcher* basati su CP nei sistemi

HPC.

Per i motivi precedenti, lo scopo di questo lavoro è quello di proporre una classe di *job dispatcher* basati su CP che siano più adatti ai sistemi HPC che eseguono applicazioni moderne. Per identificare il tipo di lavoro, proponiamo un metodo per prevedere la durata dei *jobs*, che consente di includere più informazioni *on-line* nei *job dispatcher*. I *job dispatcher* che proponiamo possono ridurre significativamente il tempo necessario per generare decisioni di invio *on-line* e possono fare un uso efficace delle previsioni di durata jobs per diminuire timeout e loro slowdown, in particolare per workloads dominati da jobs di breve durata .

Dedicada a mi familia y amigos.
Especialmente a Francisca por todo su
apoyo durante este largo proceso.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Nowadays, scientific research is enhanced by the processing of large volumes of data or heavy computations. Most of these activities are carried out in High-Performance Computing (HPC) systems, which are complex machines composed of many interconnected and independent processing units to allow a greater computation through different software techniques and/or hardware. Such systems allow studying complex problems, which in traditional computers are not feasible to be executed, such as physico-chemical simulations of drugs at the atomic level through heavy simulations, or genetic studies through large calculations, or big data analytics through mass data processing. However, there are still problems that require a higher computation magnitude, and it will be possible only to solve them on exascale systems, This means that current HPC systems require a 50-fold increase in their performance, together with an improvement in their energy efficiency [10]. The progress in hardware design is the major contributor towards these goals, including more computing nodes to achieve higher computational power, where each computing node is accompanied by newer, numerous and powerful processing units to achieve a high computational power. This is witnessed in Figure 1.1, which shows the size of the TOP500 HPC systems[1]. This ranking collects information about outstanding HPC systems around the world, and rank the systems based on the High-Performance Linpack (HPL) benchmark. Although this trend had changed in the last years because some HPC facilities have been bet for heterogeneous computing architectures. Thus, newer systems have a fewer number of computing nodes in order to reduce electrical consumption which in turn maintain or, even better, increase the computing power. Such architectures include extra computing resources different from CPU-core units, such as GPU and MIC accelerators, which additionally are efficient to execute specific floating-point arithmetic operations that are mainly used in different HPC application scenarios. Indeed, as is depicted in Figure 1.1, the first two HPC systems of the TOP500 ranking, both having a computing heterogeneous architecture, have less computing

[1]TOP500 Supercomputers sites: https://www.top500.org/

nodes as the third system, which has a CPU-based computing architecture. This means that more computing power is delivered by heterogeneous nodes than the homogeneous ones. Still, the number of nodes is high, 73% of the HPC systems in the ranking have between 500 and 10,000 computing nodes. The rest of the increase of the performance has to come from software techniques used in the computing processes, and their management together with the computing resources. Indeed, the management of the computing process and computing resources is not trivial and inefficient management may impact the performance of a system.

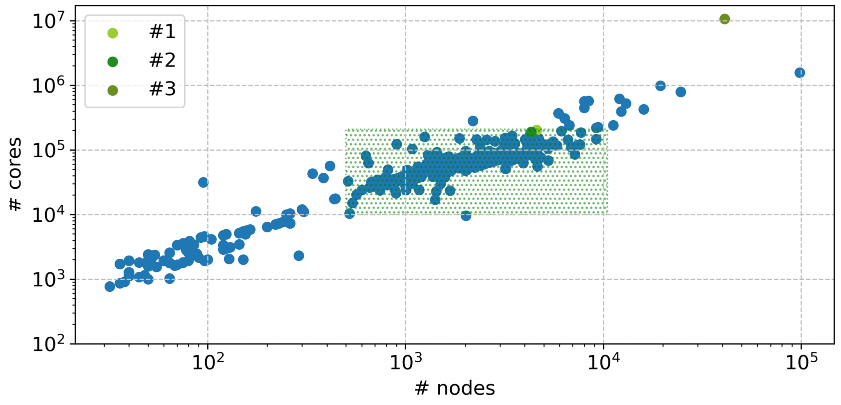


Figure 1.1: Size of TOP500 HPC systems.

In an HCP context, the execution of a computing process is known as a *job*, and corresponds to a set of instructions that allows running an (unattended) computing software on an HPC system. If a job is as a serial program, it runs on a specific node in a unique CPU-core. Instead, if a job is parallel, it may run on different “portions” of the system, and in behalf of the user requirements, it can run on thousands of processors, and use use co-processors, terabytes of memory, and storage system through a high-speed network. In general, since an HPC system is a big machine composed of numerous physical parts, and it needs to be carefully orchestrated to execute numerous jobs at the same time. The management software in charge of orchestrating jobs and resources is known as *Workload Resource Management* system, and the fundamental component that manages jobs is known as the *job dispatcher*. A job dispatcher decides which jobs to run next among those waiting in the queue (*scheduling*) and on which resources to run them (*allocation*), and this problem is known as the on-line job dispatching problem. Ideally, *dispatching decisions*, i.e. solutions to different instances of the problem, should complete all jobs in the shortest amount of time possible while keeping the system utilization high. Thus this component become critical for keeping system utilization high while keeping *waiting times* low for jobs that are competing for HPC system resources. So, a dispatching decision can achieve a certain level of Quality of Service (QoS) by handling multiple issues at the time, such as the administrator rules, system size, usage limits, software limitations, etc. Therefore, also distinct QoS levels may be reached with different dispatchers and goals.

In general, many dispatching methods currently employed in HPC systems, theoretically assume perfect knowledge of the system and its workloads. For instance, the duration of a job is unknown until it completes its execution, despite this, many dispatchers whose objective is based on the duration, rely on this attribute generating wrong dispatching decisions. Additionally, most of the *dispatching methods* used in HPC systems have their basis in traditional scheduling heuristics such as *First-In-First-Out*, *Shortest-Job-First*, even advanced algorithms, such as *Backfilling*; and suffer from making isolated decisions without considering the allocation aspect because the resource allocation is performed as a secondary process. Yet, the exact duration of a job is crucial for generating dispatching decisions to guarantee high QoS levels. Dispatchers often use the expected job duration, which is the maximum time a job is allowed to execute on the system. In the above mentioned dispatchers, the expected duration is the default value assigned by the system, which is typically the default wall-time of the queue where the job is submitted, unless the job owner supplied her own expected duration. Even in the latter case, however, users tend to use the maximum wall-time and user estimations are acknowledged to be overestimated in general [47, 34, 79]. A dispatcher that relies on overestimated durations is likely to schedule fewer jobs than possible at dispatching time, and consequently, is likely to cause unnecessary delays.

Even, if a queued-based scheduling integrates the actual duration of jobs still may not generate (near-)optimal dispatching decisions. Because, queued jobs are considered by such dispatchers one at a time, and generate uncoupled decision generating additionally fragmentation of the system. To tackle this issue, it is necessary to consider the entire queue at once, and generate a dispatching decision that satisfies the system stakeholders' expectations. This can be done using optimization-based dispatching, because dispatchers generate a dispatching decision based on a plan considering the present and future of the system, so all queued jobs will be scheduled between the current time and some time horizon. Every submitted job is planned immediately, and if a running job ends before it was estimated to end, a new dispatching decision is generated.

While the on-line job dispatching problem in HPC systems is NP-hard [14], it can be formulated as a *job scheduling and resource allocation problem* for which Constraint Programming (CP) has produced good results throughout its history [7]. The first CP-based HPC dispatcher with job waiting times as a measure of QoS was introduced in [9] and shown to obtain better solutions compared to a *Priority Rule-Based* (PRB) dispatcher [26, 62], which is widely adopted in commercial HPC workload management systems such as Altair PBS Professional [3] and SLURM Workload Manager [113]. The dispatcher was later embedded as a plug-in within the software framework of PBS professional [25]. Subsequently, another CP-based dispatcher with a similar measure of QoS with the additional feature of limiting system power consumption was presented in [18, 17] and proved to outperform a PRB dispatcher on the instances with tight power capping values. Despite the potential of these state-of-the-art

CP-based job dispatchers, certain limitations hinder their adoption for modern HPC systems, specially in the first dispatcher, such as inflexibility to deal with *heavy workloads* — workloads where resource requests greatly exceed available resources, and the CP-model is dependant on the system architecture, which causes undesirable increments in the required time by this dispatcher in generating a dispatching decision. Similar to the majority of HPC dispatchers, the expected duration corresponds to either the default wall-time or the user expected duration, which as we already said may schedule fewer jobs than possible at dispatching time, and consequently, to cause unnecessary delays.

Since HPC systems are increasingly being used for modern applications such as big data analytics and predictive model building. The jobs that run these applications correspond mainly to short-duration jobs [102]. Thus to identify such class of jobs in these application scenarios is important because HPC systems need to execute a large number of short jobs quickly. In addition, the decisions must ensure high QoS levels and meet demanding timing requirements in the on-line context, so as to minimize both waiting times and *slowdown* (the ratio between the total job duration including waiting time and the actual job duration during runtime). These measures of QoS are particularly important when HPC systems are used to provide real-time services, such as big-data visualization [106, 133, 94], where response times are critical for acceptable user experience. Prediction of actual runtime durations using simple heuristics or more sophisticated machine learning techniques is an active area of research [129, 52, 42]. Recent studies show that the use of job duration predictions when generating dispatching decisions can substantially improve QoS levels in backfilling-based dispatchers [128, 52, 42], which, in general, can be applicable to any dispatcher which bases its decisions on an important attribute such as the job duration.

1.1 Motivations

Heuristic-based dispatchers used in HPC systems may not generate (near-) optimal solutions nor cover many aspects of the problem. Thus CP-based dispatchers have been proposed to address these issues as a whole. Yet, the state-of-the-art CP-based job dispatchers are unable to satisfy the challenges of on-line dispatching. As reported in [25], the first CP-based dispatcher is not resilient to heavy workloads, where resource requests greatly exceed available resources. The time spent by this dispatcher in generating a dispatching decision increases dramatically as more jobs requiring high system utilization arrive to the system. Another issue is the difficulty to scale properly to big systems (which includes hundreds to thousands of computing nodes), which are the most common HPC systems architectures. This dispatcher models the on-line job dispatching problem in a unified model, that is, considers the scheduling and allocation problem in a *single CP model*, however, it may not scale well on big systems due to how its decision variables are modeled. The number of decision variables depends on the number of queued jobs and all of their possible allocations on each computing node, thus the model of this dispatcher is highly dependant on the system size, the availability and the request of resources, which we detail later in Chapter 2.3. In an effort, to improve the previous issues, a hybrid dispatcher [18, 17] is proposed. However, it was initially employed in off-line mode [18], and later also in on-line mode [17] but on workloads of maximum 1,000 jobs submitted in a time window of half an hour. A more realistic scenario where jobs arrive continuously and many of them end up waiting in a queue due to unavailable computational resources increases greatly the difficulty of generating dispatching decisions. In addition, these dispatchers use the expected duration as the default value assigned by the system, which is typically the default wall-time of the queue where the job is submitted, unless the job owner supplied her own expected duration. Therefore, such dispatchers may schedule fewer jobs than possible at dispatching time causing unnecessary delays. These limitations jeopardize achieving high QoS levels, and consequently impede the adoption of CP-based dispatchers in HPC systems.

We believe CP-based dispatchers should be adapted to generate high-quality dispatching decisions independently of the architecture and size of the systems, specially CP-based dispatchers that model the scheduling and allocation problems together in a unified model, because they may incorporate additional restrictions, such as power management in the broad sense (power capping, switching idle nodes, reduce power consumption base on job allocation, etc.), which is a must of future HPC systems, and could provide better dispatching decisions for current and future HPC systems.

Together with improving the models of the dispatchers, it is important to define the optimization criteria for this problem. Currently, these two state-of-the-art CP-based dispatchers uses a metric which is not job specific feature that can be decided on-line at the time of dispatching. So, such a value may not be informative on the current job submission status so as to generate a dispatching decision of high quality. Therefore, by studying different openly available workloads [2], as well as other system workloads that we had access to, we have been able to distinguish an interesting trend that can be a perfect representative attribute of jobs. This trend has been mentioned in various studies [112, 101, 102] but that has not taken enough advantage to improve the performance of HPC systems. We already know that workloads of current HPC systems tend to be a mix of many jobs that run for less than one hour, with fewer longer jobs, resulting in a heavy-tailed job duration distributions [102].

Figure 1.2 shows this clear trend in more than 12 HPC systems used in different parts of the world, America, Asia, and Europe; more than 60% of the jobs last less than 1 hour of these systems. Indeed, due to recent developments in big data analytics and new programming paradigms such as map-reduce, put further emphasis on short jobs, with jobs being split into tasks with inter-dependencies [32]. Hence, in these application scenarios, HPC job dispatchers need to rapidly process a large number of short jobs in making on-line decisions

[2]Parallel Workloads Archive `https://www.cse.huji.ac.il/labs/parallel/workload/logs.html`.

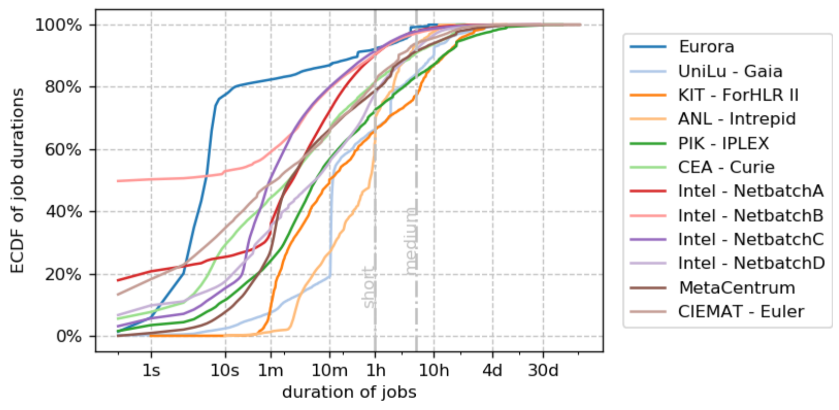


Figure 1.2: Distribution of durations of jobs on different HCP systems.

so as to improve the QoS metrics. Waiting so long for the execution of a short job is not well perceived by users, thus this class of jobs should be treated with higher priority. An interesting metric to consider given this observation is the job slowdown, which is the normalization of the waiting time. The job slowdown has been considered as a good QoS metric since it matches the user expectations that a job's response time, which is the time that passes between the submission time and the completion time, will be proportional to its running time. Indeed, 50 years ago, in [60] was suggested that job slowdowns should be used to prioritize jobs for scheduling. Thus, we propose the use of job slowdown as optimization criteria, where the objective function will be the minimization of the slowdown of queued jobs. However, using the job slowdown requires knowing the duration of the job, that we already mentioned is not available, instead, only an upper bound of this value is known. In addition, identifying the actual duration of jobs is not a straightforward task because users, in general, tend to overestimate the expected duration of jobs. Although a user repeats a job with the same or similar features, and she already knows an estimated duration based on the previous one, she, typically, will still request more time than the required one [47, 34, 79]. Since duration is an important asset in job scheduling, different prediction methods have been proposed in the literature, from a simple average of the duration of the last jobs to complex support vector machines that try to find a correlation between distinct job features [121]. Our motivation is to study whether transforming the log data produced by an HPC system into useful knowledge about its workload may produce better dispatching decisions. Given that HPC systems produce large amounts of data in the form of logs tracing resource consumption, errors and various other events during their operation. Data science can transform this raw data into knowledge through models built from historical data capable of anticipating unseen or future events. We believe that predictive computational models obtained through data-science tools will be indispensable for the operation and control of future HPC systems. Therefore, relying on predictive models to obtain useful knowledge about the workload from the log data of an HPC system can be a good option to increase the knowledge of the workload, with the purpose of making better dispatching decisions. Given that we want to use the job slowdown to priorize short jobs, we focus on job duration predition.

One of the challenges of job dispatching research is the intensive experimentation necessary for evaluating and comparing dispatchers in a controlled environment because using an actual HPC system for experimenting is not realistic. Therefore, simulating a WMS is essential for conducting controlled dispatching experiments. Such a simulator must simplify the evaluation of multiple dispatchers across multiple workloads with different formats, which may contain thousands to millions of jobs. If workloads contain millions of jobs, it must handle the workload efficiently to avoid affecting the performance of the dispatcher. Thus, a simulator that efficiently can process heavy workload dataset with a low memory footprint, fast processing, and easy customization is a must in dispatching research. However, existing simulators lack of different features that we require to conduct our experiments.

1.2 Goals

The main goal of this dissertation is to propose CP-based dispatchers that are capable of processing heavy workload datasets of systems of any size and take advantage of job duration prediction methods to achieve high Quality of Service levels. To achieve this main goal, it is required to propose adaptations to the model and search control mechanisms of state-of-the-art CP-based dispatchers so as to make them resilient to heavy workloads and applicable to on-line dispatching. Additionally, we reuse these mechanisms in a new class of CP-based dispatchers which are composed of fewer decision variables than the utilized in the model of the pure state-of-the-art CP-based dispatchers and do not depend proportionally with the size of the system. We believe such models due to this independence may scale better to big systems.

Given that we want to improve the quality of dispatching decisions in workloads that runs modern applications, which are mainly composed of short jobs, it will be required to identify such a class of jobs. So, as a secondary goal, we study and develop a job duration prediction method, which can be integrated with any dispatcher to provide useful information regarding the estimated job duration, so as to improve the quality of the dispatching decisions. Another secondary goal is the development of a simulator of a Workload Management System which will easy the simulation task by defining the proper framework to carry out our job dispatching research activities.

Thus, based on our goals and motivations the thesis statement of this dissertation corresponds to:

> *Jobs and resources of HPC systems can be efficiently and effectively managed using job dispatchers based on the Constraint Programming paradigm, independent of the size and workload density of the system.*

This research work is based on simulation on a restricted number of workload logs of actual systems, therefore dispatchers and predictors presented here must be tested on the resource configuration and using an actual workload dataset of the target system before putting in them in production. This research used only one Constraint Programming system for all the experimental study, therefore the performance of the models presented here exclusively depends on the experimental settings.

1.3 Contributions

The main contribution of this dissertation is to establish CP-based dispatchers as a choice for current HPC systems. However, the state-of-the-art CP-based job dispatchers are unable to satisfy the challenges of on-line dispatching and take advantage of job duration predictions. Thus, we propose a class of CP-based dispatchers that take these considerations to be suitable for HPC systems running modern applications. The proposed dispatchers are able to reduce the time required for generating on-line dispatching decisions significantly, and are able to make effective use of job duration predictions to decrease waiting times and job slowdowns, especially for workloads dominated by short jobs. In spite of the improvement presented with the new class of dispatchers, there are still issues regarding big systems for the pure CP-based dispatchers, that is, CP-based dispatchers that model the scheduling and allocation problem as a whole. We focus only on pure CP-based dispatchers because such dispatchers can handle more restrictions meanwhile solve the on-line job dispatching problem. Indeed, we believe that such a class of dispatcher will be necessary to cope with new challenges on actual HPC systems, such as power consumption restrictions, the anticipation of failures of resources, among others. So, we propose a new class of pure CP-based dispatchers which has to scale well on big systems and also provide significant results on the small and medium systems.

A part of the main contribution was published in [50].

To achieve our main contribution, this dissertation also provides secondary contributions:

AccaSim, a Workload Management System simulator for job dispatching study This simulator resulted in an open-source library, implemented in Python, which is freely available for any major operating system, and works with dependencies reachable in any distribution. AccaSim is scalable to large workload datasets and provides support for easy customization, allowing to carry out experiments across different workload sources, resource types, and dispatching algorithms. Moreover, AccaSim enables users to develop novel advanced dispatchers by exploiting information regarding the current system status, which can be extended for including custom behaviors such as energy and power consumption and failures of the resources. Thus, AccaSim can be used to mimic any real system, including those possessing heterogeneous resources, develop advanced dispatchers using for instance power and energy-aware, fault-resilient algorithms, and test and evaluate them in a convenient way over a wide range of workload sources by using real workload traces or by generating them. This work was published in [48, 49].

A data-driven job duration prediction We propose two heuristics that construct job profiles from the available workload data. For the first heuristic, we identified different attributes that can be useful to classify similar jobs, and then we predict the job duration based on specific similarity rules. The second heuristic maintains the same ideas that in the first one, and, besides, introduces a 'rewarding methodology' to deal with users that request the wall-time accurately. These heuristics are very simple; thus, they do not produce any overhead during the prediction and yet proved to be very effective on job dispatchers that actively use the wall-time as the expected job duration. A part of this work was published in [51].

1.4 Organization of the dissertation

The rest of this dissertation is organized as follows.

Chapter 2, Background. We give an overview of the main concepts addressed in this dissertation. We focus on the High-Performance Computing (HPC) concept, with a special emphasis on the on-line job dispatching problem. Next, we introduce the Constraint Programming (CP) paradigm because most of the proposals presented in this dissertation are based on this technology, so we explain how it works and how problems are solved. Finally, we formally introduce state-of-the-art CP-based dispatchers on which this proposal aims to improve or come up with new models.

Chapter 3, Accasim: A Workload Management System simulator. Since it is required to execute dispatchers under the same conditions, and the available tools were not able to customize as needed, we developed a necessary tool to conduct our experiments. This tool was used to carry out all the experiments which require job submission simulation through this dissertation.

Chapter 4, Job duration prediction in HPC systems. The job duration is a valuable asset to make efficient dispatching decisions, however, this value is unknown and users tend to overestimate this value which affects duration-based decisions. To tackle this issue, we propose a job duration prediction method aimed to improve well-known dispatchers without interfering with their main algorithms.

Chapter 5, Active usage of job duration prediction on scalable to heavy workloads CP-based dispatchers for HPC systems. Since state-of-the-art CP-based dispatchers, especially the dispatcher that models the allocation and scheduling problems together, a pure CP-based dispatcher, do not scale well on some dispatching instances we propose some significant improvement to their model and search control. In addition, we propose an active integration of job duration prediction to improve the quality of service of HPC systems.

Chapter 6, Towards system size-independent CP-based dispatchers for HPC systems. Although in the previous chapter we made CP-based dispatchers scalable to heavy workloads, still there are some issues related to the size of systems, which has an impact on the performance of the pure state-of-the-art CP-based dispatchers. We focus on such a dispatcher class, i.e. with a model that considers both the allocation and the scheduling problem as a whole, because it can incorporate better additional constraints to deal with complex scenarios than hybrid dispatchers, which decision may be overridden due to incompatibilities. Thus, we propose two new pure models which are less and completely independent of the size of the systems.

Chapter 8, Conclusions and future work. We present our conclusions of this dissertation and describe future work.

Chapter 2

Background

Before presenting contributions, we introduce in this chapter key concepts related to this dissertation. We divided this chapter into three principal sections. In Section 2.1, we introduce the key concepts of High-Performance Computing to know the operations of such systems and to witness of the needs that they require to achieve a high Quality of Service by handling efficiently the on-line job dispatching problem. Next, in Section 2.2, we introduce the key concepts of Constraint Programming, which are required to understand how the proposed dispatchers work. Finally, in Section 2.3, we describe the state-of-the-art CP-based dispatchers used to tackle the on-line job dispatching problem in HPC systems.

2.1 High performance computing

High-performance computers (HPC), also known as Supercomputers, and their usage are part of the current trends of the last technological improvements, playing a crucial role in doing research and improving the business capabilities thanks to their vast computing power. This modern computing has been powered by continuous and significant technology advances and achieved through innovations in computer architecture, programming models, and needs of end-user goals that could only be addressed by computational means. Classical scientific and engineering problems, which are usually studied on computers, has been extended across all branches of industry, commerce and, government thanks to the high processing power of HPC. Covering such areas generate an impact on almost every aspect of daily life: energy, transportation, communications, medicine, infrastructure, finance, business management, and the manufacture of both new and traditional consumer products [41]. In general, HPC systems have become fundamental tools to solve complex, compute-intensive, and data-intensive problems, enabling new scientific discoveries, innovation of more reliable and efficient products and services, and new insights in an increasingly data-dependent world. This can be witnessed for instance in the annual reports[3] of PRACE and the recent report[4] by ITIF which accounts for the importance of HPC to the global economic competitiveness.

HPC is a complex set of technologies that involves multiple disciplines of the Information and Communications Technology domain to take advantage of the computing hardware using the available software to solve complex computational problems through computer modeling, simulation, and data analysis. A problem is defined as a software application, which runs during a specific time on such a system, and once it completes its execution, it gives answers to research questions. How fast a problem can be solved depends on how complex it is and how fast is the machine where the application is executed (and how good it is coded).

Thanks to the improvement in the technology, architecture, programming techniques, algorithms, and system software, the computing power has been increased over the years with an exponential growth, passing from thousands of basic operations during the 40s to over 100 petaFLOPS [5] in the last years[6]. A FLOPS is a performance metric which measures "floating-point operations per second" and is widely used to compare HPC systems. A widely used HPC benchmark is the "highly parallel Linpack" (HPL), which solves a set of linear equations in dense matrix form, which are selected by the HCP community.

Since 1993, TOP500 gather the HPL benchmark results to compare the most outstanding HPC systems. From the available data, a simple conclusion emerge, the performance is incremented almost 90% each year. So, HPC systems may reach the exascale performance during this year (2020) according to TOP500, if the development continues its exponential growing, as depicted in Figure 2.1 (Source: `https://www.top500.org/statistics/perfdevel/`). The last update of the fastest HPC systems, corresponding to June 2019, shows as Summit in the first place. Summit, in Figure 2.2 is an IBM-built supercomputer housed at the Department of Energy's (DOE) Oak Ridge National Laboratory (ORNL), which has a performance of 122.3 petaflops on the HPL. This supercomputer has 4,356 nodes, each one equipped with two 22-core Power9 CPUs, and six NVIDIA Tesla V100 GPUs. The nodes are linked together with a Mellanox dual-rail EDR InfiniBand network.

In general, an HPC machine appears as rows upon rows of many racks taking up thousands of square feet and consuming potentially multiple megawatts of electrical power. An increase of at least one order of magnitude of computing power is also translated in a higher power consumption, 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.

An HPC system has basic functionality and subsystems in common with the personal computer, however its organization, interconnectivity, and scale of the component resources and the ability of the supporting software to manage the operation of the system at a high degree of logical and physical parallelism. A

[3]`http://www.prace-ri.eu/praceannualreports/`

[4]`http://www2.itif.org/2016-high-performance-computing.pdf`

[5]petaFLOPS corresponds to 10^{15} Floating Point Operations Per Second

[6]According to the data of TOP500 `http://www.top500.org`.

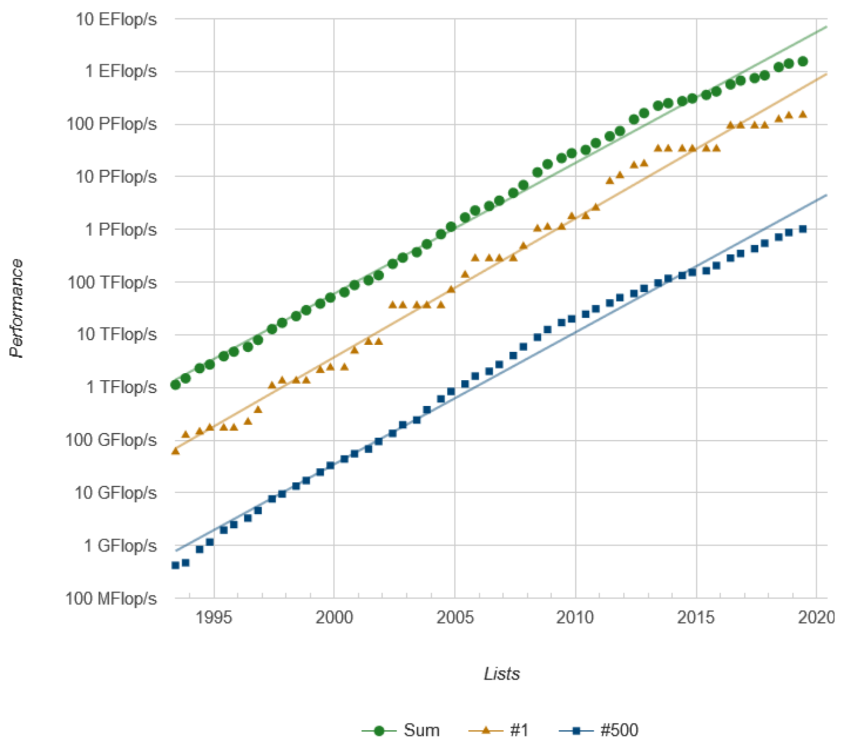


Figure 2.1: TOP500: Projected performance development of HPC systems.

CPU socket in a personal computer incorporates some parallelism, whereas an HPC system is structured in far more levels with specific methods to coordinate and solve a shared problem.

In spite of HPC systems are composed of many nodes, sometimes this quantity is not enough to cover the users needs. When two or more requests are made at the same time, or are in the queue at the same time, to be serviced by the same single resource, either hardware or software, only one can proceed. The other(s) must wait until the first request is retired and the required resource is freed. If this postponing is extended in time, taking longer to process the entire queue. In addition, this has a cascading effect on the following requests, and some resources may be blocked and its potential capability wasted for the duration of the delay. Such events occur unpredictably and create uncertainty during the execution.

2.1.1 Workload Management Systems

A Workload Management System (WMS), also known as Resource and Job Management System (RJMS), resource and task manager, or batch scheduler, is a middleware software which in general manages jobs and resources, by check-

Figure 2.2: IBM Summit HPC system at Oak Ridge National Laboratory, United States of America.

ing, monitoring, profiling, accounting job submissions and system utilizations in order to distribute the available resources to jobs. To do so, a WMS incorporates a scheduling and allocation algorithm to choose where and when jobs will run. From a point of view of resources management, a WMS is responsible to collect and provide all information concerning the system resources. This information has to be available to the dispatcher to initiate the job dispatching and to the user or administrator to inform about the availability and the state of the system. From a point of view of job management, a WMS provides the means for definition, submission, and monitoring of jobs. Considering both points of view emerges the dispatcher. Its main role is to assign jobs according to the users needs and predefined rules and policies, upon available computational resources that match the demands.

Well-known WMS are:

- PBS Pro [7]
- SLURM [8]
- OAR [9]
- Torque [10]
- Flux [11]
- Condor [12]
- LSF [13]

[7] `http://www.pbsworks.com`
[8] `http://slurm.schedmd.com`
[9] `https://oar.imag.fr/`
[10] `https://www.adaptivecomputing.com/products/torque/`
[11] `https://arc-ts.umich.edu/flux/`
[12] `https://research.cs.wisc.edu/htcondor/`
[13] `https://www.ibm.com/us-en/marketplace/hpc-workload-management`

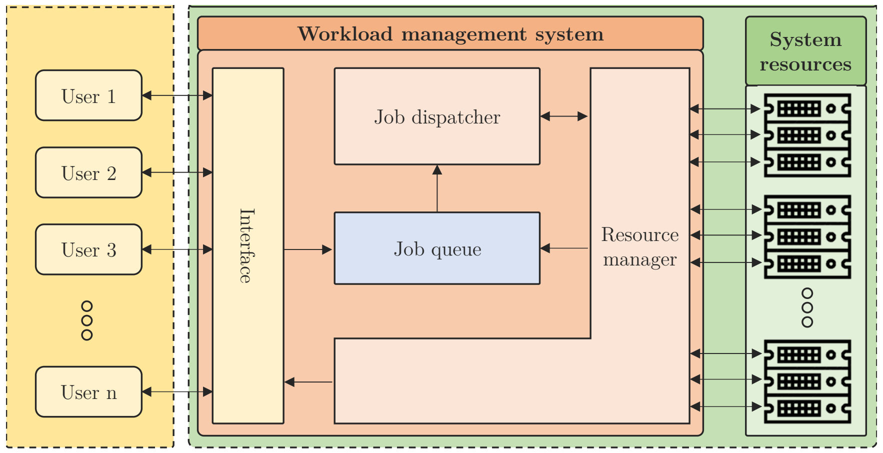


Figure 2.3: WMS in an HPC system.

Therefore, a WMS is an important software of an HPC system, being the main access for the users to exploit the available resources for computing. A WMS manages user requests and the system resources through critical services. A user request consists of the execution of a computational application over the system resources. Such a request is referred to as job and the set of all jobs are known as workload. The jobs are tracked by the WMS during all their states, i.e. from their submission time, to queuing, running, and completion. Once a job is completed, the results are communicated to the respective user. Figure 2.3 depicts a general scheme of a WMS.

A WMS offers distinct ways to users for *job submission* such as a GUI and/or a command line interface. A submitted job includes the executable of a computational application, its respective arguments, input files, and the resource requirements. An HPC system periodically receives job submissions. Some jobs may have the same computational application with different arguments and input files, referring to the different running conditions of the application in development, debugging and production environments. When a job is submitted, it is placed in a *queue* together with the other pending jobs (if there are any). The time interval during which a job remains in the queue is known as waiting time. The queued jobs compete with each other to be executed on limited resources.

We recall that a *job dispatcher* decides which jobs waiting in the queue to run next (*scheduling*) and on which resources to run them (*allocation*) by ensuring high system utilization and performance. The dispatching decision is generated according to a policy using the current system status, such as the queued jobs, the running jobs and the availability of the resources. A suboptimal dispatching decision could cause resource waste and/or exceptional delays in the queue, worsening the system performance and the perception of its users. A (near-)optimal dispatching decision is thus a critical aspect in a WMS.

The dispatcher periodically communicates with a *resource manager* of the WMS for obtaining the current system status. The *resource manager* updates the system status through a set of active monitors, one defined on each resource which primarily keeps track of the resource availability. The WMS systematically calls the *dispatcher* for the jobs in the queue. An answer means that a set of jobs are ready for being executed. Then the dispatching decision is processed by the *resource manager* by removing the ready jobs from the queue and sending them to their allocated resources. Once a job starts running, the *resource manager* turns its state from "queued" to "running". The *resource manager* commonly tracks the running jobs for giving to the WMS the ability to communicate their state to their users through the interface, and in a more advanced setting to (let the users) submit again their jobs in case of resource failures. When a job is completed, the *resource manager* turns its state from "running" to "completed" and communicates its result to the interface to be retrieved by the user.

2.1.2 The on-line job dispatching problem in HPC systems

Since a CPU-core can only do one thing at a time and one of the goals of high-performance computing is to reduce the time for computations, jobs must be managed correctly. Therefore a job dispatcher must ensure that each job has dedicated access to the resources it needs, by deciding the start time of submitted jobs and the resources on which the jobs will be allocated, by providing an efficient dispatching plan which covers all users' needs and obeys the system rules. This process is known as job dispatching, job scheduling and allocation, or simple job scheduling, and it has been studied for a long time with the aim of improving the system performance [45, 134, 65, 108, 70]. From a optimization point of view this process is modeled as a dispatching problem, also named as scheduling and allocation problem, which has been widely studied [35, 21, 22, 99] given the challenges of an NP-hard problem [14].

However, some discrepancies are observed between a (off-line) job dispatching problem applied to an on-line context such as the on-line job dispatching problem in HPC systems. In an off-line dispatching problem, it is assumed that jobs and their requested resources are known beforehand, as well as the available resources to allocate those jobs, therefore both elements in the problem are static. Nevertheless, in an on-line context, there is a much higher uncertainty. During unknown times, a system may receive job submissions from HPC users. After some events are triggered (specific time intervals, after a jobs end and there are queued jobs, or every time a job is queued, among others), the WMS calls the job dispatcher to generate a dispatching decision. During such a call, the WMS provides information about current running jobs together with their allocated resources, in addition to the queued jobs to be dispatched with their requested resources. There is a possibility that the system may not have enough availability to allocate all queued jobs, so only a subset of them will be dispatched, the remaining ones will be re-queued until the dispatcher is called again. The previous workflow is repeated unlimited meanwhile the system is on and accepting job submissions.

A *dispatching decision* (solution to an specific instance) is obtained according to a policy using the current system status, such as the queued jobs, the running jobs and the availability of the resources. However, the overall system performance cannot be determined by a single dispatching decision, we emphasize this problem occurs on-line, i.e., there are many dispatching problems during the lifetime of an HPC system. Further, if a dispatching decision of an instance is to dispatch a subset of jobs, different dispatchers may generate distinct solutions, so later instances during the system operation and may show distinct performances.

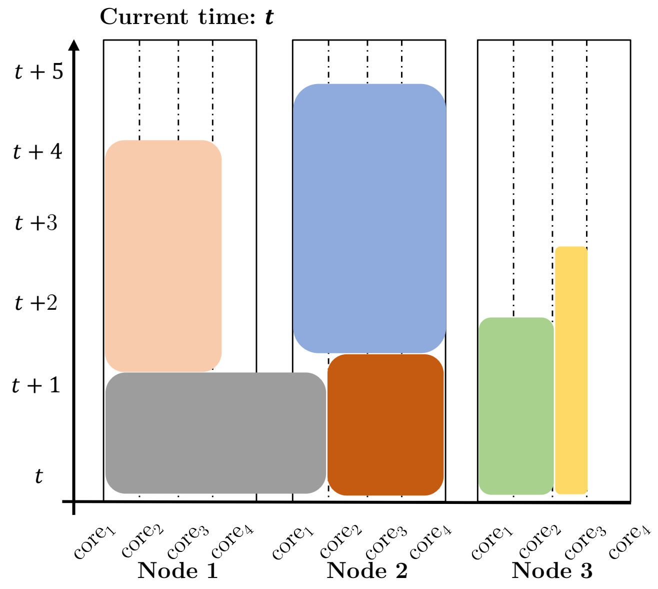


Figure 2.4: A dispatching decision.

Analyzing the on-line job dispatching problem in HPC systems, we realize that jobs are the main actors in the problem. Considering the two types of jobs, queued and running, and knowing the capacity of the system, it is possible to establish the consumed and the available resources, where the last ones are key to dispatch the queued jobs. As we mentioned in the previous section, a job is only a user request, which consists of a script to execute a computational application over some system resources. From an optimization point of view, there is another important attribute of jobs, which is the expected job duration and corresponds to the maximum time a job is allowed to execute on the system. The expected duration is the default value assigned by the system, which is typically the default wall-time of the queue where the job is submitted, unless the job owner supplied her own expected duration. Figure 2.4 illustrates how jobs (the colored boxes) can be allocated to certain resources (cores on different nodes) of a HPC system at certain times.

Formally speaking, the *on-line dispatching problem* in an HPC system takes place at a specific time t for the queued jobs Q. A typical HPC system is composed of N nodes, with each node $n \in N$ having a capacity $cap_{n,r}$ for each of its resource type $r \in R$, giving the total amount of available resource. The system may not be completely available, because some resources may be already allocated to the running jobs in G during some intervals.

Each job $i \in Q$ has the arrival time $q_i \leq t$ to the queue, which is unknown before the arrival, and a demand $req_{i,r}$ giving the amount of resources required from r. Each job $g \in G$ has been allocated previously to some resources, therefore those resources are already occupied and cannot be shared with other jobs. Allocated resources of a job g are freed once the execution of g ends, $s_g + d_g^r$ where s_g is the starting time and d_g^r the runtime or actual duration. Once a running job end, the WMS marks the occupied resources as available again.

The *on-line dispatching problem* at time t consists in *scheduling* each job i by assigning it a start time $s_i \geq t$, and *allocating* i to the requested resources during its expected duration d_i, such that the capacity constraints are satisfied: at any time in the schedule, the capacity $cap_{n,r}$ of a resource r is not exceeded by the total demand $req_{i,r}$ of the jobs i allocated on it, taking into account the presence of jobs already in execution G.

Once the problem is solved, only the jobs with $s_i = t$ are dispatched. The remaining jobs with $s_i > t$ are queued again with their original q_i. It is the workload management system software that decides the dispatching time t and the subsequent dispatching times. A typical objective is to minimize the sum of the waiting times $s_i - q_i$, which is a metric that can be easily perceived by HPC users.

In actual HPC systems, the dispatching problem is mainly addressed by heuristics methods to find feasible solutions, instead of optimal ones, even though a (near-)optimal solution is a critical requirement in HPC systems. A sub-optimal dispatching decision solution could cause exceptional delays in the queue, affecting the QoS. Most WMS include the well-known FCFS (First Come First Served) algorithm, where jobs are process in order as they arrive in the queue until a job cannot be started. Other dispatchers based on the user expected duration are also available, such as Short Job First, or more advanced dispatchers like backfilling scheduling algorithms [119]. We give an overview of those algorithms in Section 2.1.3.

In general, dispatchers based on systematic complete search methods, such as Constraint Programming (CP), are not well accepted in actual systems, given the required time to generate a dispatching decision or to scale properly to big systems. Despite this, we believe such dispatchers will be necessary to cover many and different aspects during a dispatching decision, given that CP takes advantage of the integration of local consistency and propagation techniques inside the search algorithm. We describe state-of-the-art CP-based dispatchers in Section 2.3.

2.1.3 Job dispatchers in HPC systems

The purpose of a job dispatcher is to reserve resources for users' jobs to ensure jobs run at their highest performance. It keeps a given compute node from being overloaded, and places jobs on hold until resources are available. So, the job dispatcher must decide when and where execute each submitted job. Since it is a hard problem [14], most of the proposed solutions are heuristic-based methods, which are fast but do not guarantee optimality.

In addition, such methods are a combination of scheduling and allocation methods, which together generate a dispatching decision. In a progressive manner, queued jobs are sorted by the scheduling method and, then in that order jobs are allocated, however, a selected job may not be always allocated to the requested resources, so, the dispatching method, as a whole, decides to continue or stop processing the queue.

We examine five scheduling methods and two allocation methods reported in the literature. In the following, we give intuitions for the algorithms underlying these scheduling methods:

First In, First Out First In, First Out (FIFO) dispatch jobs in the order in which they enter the queue. This is a very simple strategy to implement, and works acceptably in system with a low job load.

Shortest job first, longest job first Shortest Job First (SJF) and Longest Job First (LJF) use the estimated duration at dispatching time, sorting all jobs that have to be dispatched in ascending (or descending) order, and then mapping the shortest job (or the longest job) to a resource [124]. Both algorithms continue moving through the sorted list until no available resources remain for allocating to the current job. The aim of SJF is to reduce the waiting time of the short jobs, thus causing delays for the execution of the long jobs. Conversely, LJF reduces the waiting time of the long jobs, causing slowdown for short jobs.

EASY-Backfilling A key element of many commercial dispatchers is the *backfilling* algorithm [136] which starts scheduling jobs stepping through a priority list such as FIFO, SJF or LJF. If a job cannot be dispatched due to lack of available resources (blocked job), backfilling calculates the time in the future when enough resources will be released to run the blocked job, based on the estimated duration of running jobs. While the blocked job is waiting, the dispatcher maps other jobs in the queue over the available resources. If, however, the durations have been underestimated, the resources for the blocked job will not be available when needed, which can force termination of the running jobs. In such a case, EASY-Backfilling (EBF) [136] does not terminate the running jobs but instead delays the starting time of the blocked job.

Priority rule-based As an extension of the first-in-first-out policy, many dispatchers sort the set of jobs to be scheduled by certain priority, running those with higher priorities first. This algorithm is referred to as Priority Rule-Based (PRB) [62, 26] and is widely used in commercial HPC dispatchers[14,15].

Next, we describe the idea of the allocation methods:

First Fit First Fit or *all-requested-computers-available* policy for resource allocation [136]. For each scheduled job, this policy searches sequentially the nodes in an attempt to find resources available for running the job, and if succeeds, it maps the job onto those nodes.

Best Fit Best Fit sorts the nodes in non-decreasing order of the number of available resources after each successful allocation, For each scheduled job, this policy searches sequentially the nodes in an attempt to find resources available for running the job, and if succeeds, it maps the job onto those nodes. Thus, it tries to fit as many jobs as possible on the same node, to decrease the fragmentation of the system

2.1.4 HPC workloads

This section contains information regarding the workloads on HPC systems used through this dissertation. As we motivated in the introduction, our work depends on workload datasets to evaluate our contributions in HPC job dispatchers. These workloads datasets have been provided (Eurora system) or are freely available online [16] with some restrictions in the data available (Gaia system, KIT system). The freely available workloads datasets are part of various parallel systems in production use in various places around the world collected and submitted by people of their own IT facilities. Since the original logs come in different formats, the administrators of the website unified the workload datasets under the Standard Workload Format (SWF) [17] [30]. The SWF in its last version (2.2) includes the following data for each job entry, the field starting with * corresponds to an optional field and may not be present in the workload dataset:

1. **Job number**: just a counter, starting from 1.
2. **Queued time**: Corresponds to the submitted time.
3. **Waiting time**: The difference between the job's submit time and the time at which it actually began to run in seconds.
4. **Duration**: It is the runtime in seconds, i.e. the wall clock time the job was running (ending time minus starting time).
5. **Number of allocated processors**: The number of processors the job uses.

[14]Altair PBS Works (`http://www.pbsworks.com/`).

[15]SLURM Workload Manager (`https://slurm.schedmd.com/`).

[16]The parallel workload archive (`https://www.cse.huji.ac.il/labs/parallel/workload/`)

[17]The Stantard Workload Format (`https://www.cse.huji.ac.il/labs/parallel/workload/swf.html`)

6. ***Average CPU time used**: The average over all processors of the CPU time used in seconds, and may therefore be smaller than the duration clock runtime.
7. ***Used memory**: The average usage per processor in kilobytes.
8. ***Requested number of processors**: The requested number of processors the job will use.
9. ***Expected duration**: The user runtime estimate.
10. ***Requested memory**: The average memory requested per processor in kilobytes.
11. **Status**:
    - **0**: Job failed.
    - **1**: Job completed.
    - **5**: Job cancelled.
12. **User ID**: Identifier of the user.
13. ***Group ID**: Identifier of the group. Some WMS control resource usage by groups rather than by individual users.
14. ***Executable (application) number**: The number of different applications appearing in the workload.
15. **Queue number**: The number of different queues in the system.
16. ***Partition number**: The number of different partitions in the systems.
17. ***Preceding job number**: The number of a previous job in the workload, such that the current job can only start after the termination of this preceding job.
18. ***Think Time from preceding job**: The number of seconds that should elapse between the termination of the preceding job and the submission of this one.

We note that workloads datasets using the SWF not consider different interesting aspects that we cover in this dissertation such as name of jobs (for a privacy issue), job allocation, more information about the job requests, etc; so experiments involving these datasets cannot be so detailed with regard of the experiments using the Eurora workload dataset. We highlight that in an actual system all attributes used in this dissertation will be available and provided at dispatching time by the WMS.

Eurora and its workload dataset The Eurora system [28] was hosted at CINECA[18], the largest datacenter in Italy, and was ranked first on the Green500 list in July 2013. Eurora had a modular architecture based on nodes (blades). Each node had 2 octa-core CPUs (Intel Xeon E5) and 2 expansion cards that can be configured to host an accelerator module. Of the 64 nodes, half of them hosted 2 powerful NVidia GPUs (Nvidia Tesla Kepler), meanwhile the other half were equipped with 2 Intel MIC accelerators (Intel Xeon Phi Knights Corner). Each node had 16GB of RAM. These 64 nodes were dedicated exclusively to computation, with the user interface was managed by a separate node. The resulting system is highly heterogeneous, making allocation of resources to jobs nontrivial. Eurora was used by scientists across Italy to perform simulation studies from different fields, hence the workload is also heterogeneous, in the sense of request of resources and durations.

The workload data includes logs for over 400,000 jobs submitted between March 2014 and August 2015. For each job, we have information on the submission, start and end times, queue, wall-time, user and job name, together with resources used and their allocation on the various nodes. The workload data was collected through a dedicated monitoring system [10].

Figure 2.5 shows the distribution of job duration of the workload. 92.15% of the jobs in the workload based corresponds to *short* jobs (under 1 hour), 7.07% are *medium* jobs (between 1 and 5 hours) and 0.78% are *long* jobs (over 5 hours). Considering these classes together, the Eurora system executed jobs during 121,885 hours in total, where the medium jobs used most of the resources, with 70.43% of the total while short and long jobs use only 8.94% and 20.63%, respectively. Hence, the workload is quite varied from this point of view. The figure demonstrates the existence of many short jobs and fewer longer jobs, with a long-tailed distribution of job duration, which is typical to HPC [112] and cloud systems [101].

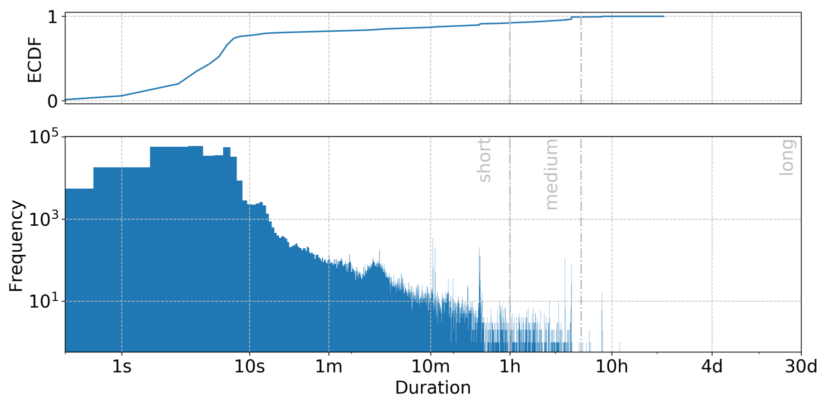


Figure 2.5: Distribution of job durations on the Eurora system.

[18]The Italian Inter University Consortium for High Performance Computing (`http://www.cineca.it`).

We further divide jobs into three classes based on the computing resources that they require: *CPU-based* jobs use CPUs only, while *MIC-based* and *GPU-based* jobs use MIC or GPU accelerators, respectively, in addition to CPUs.

Job class	Share	Count	Average duration [hh:mm:ss]
All	100%	404,866	00:18:03
CPU-based	24.56%	99,431	00:49:19
MIC-based	0.69%	2,810	00:55:42
GPU-based	76.75%	302,625	00:07:26

Table 2.1: Frequency and average duration of all jobs and the three classes CPU-based, MIC-based and GPU-based in the Eurora workload.

Table 2.1 shows statistics for each computing resource class in the workload. We observe that GPU-based jobs are the most numerous, followed by CPU-based jobs, while MIC-based jobs are relatively few. In terms of duration, we observe that CPU-based jobs are on average longer than GPU-based jobs, consuming significantly more resources. This heterogeneity of job classes increases the difficulty of allocation decisions. Since CPU-based jobs are longer, they may keep nodes that have accelerators busy for longer periods, during which their accelerators are not available for other jobs. Given that GPU-based jobs are the most frequent, this can cause bottlenecks to form in the system.

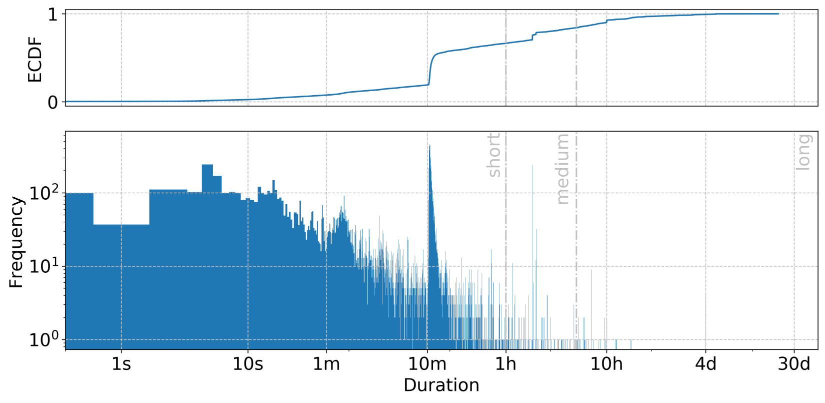


Figure 2.6: Distribution of job durations on the Gaia system.

Gaia and its workload dataset The Gaia cluster is one of the 4 clusters operated by the University of Luxembourg HPC Center [19] (ULHPC). ULHPC operates a total of 690 computing nodes for a total cumulative capacity of 1,263 TFlops (11,228 CPU cores) and a shared storage capacity of 8,742 TB

[19]`https://hpc.uni.lu`

(+ 1020 TB for backup). The Gaia system is a heterogeneous cluster that has been upgraded several times after its release in 2011. The workload dataset of the Gaia system spans from May to August 2014 with 51,987 jobs, and it was used mainly by biologists working with large data problems and engineering people working with physical simulations. The system configuration, during this period featured 151 nodes, manufactured by Bull and Dell, with a total of 2004 cores. 20 nodes had NVidia Tesla-class GPUs accelerators. Full details about its configuration and history are available in the University of Luxemburg Gaia Cluster site. [20].

The Gaia workload is composed by 66.45% of *short* jobs, 17.70% of *medium* jobs, and 15.85% of *long* jobs, as depicted in Figure 2.6. Similar to the job duration distribution of the Eurora workload dataset, the Gaia workload dataset also shows a long-tailed distribution of job duration, which enforces the idea of giving priority to jobs regarding their duration. The total CPU time employed by Gaia during these 3 months corresponds to 206,815 hours, where long jobs where used most of the resources with 86.81% of the total, whereas 9.97% and 3.21% corresponds to medium and short jobs, respectively.

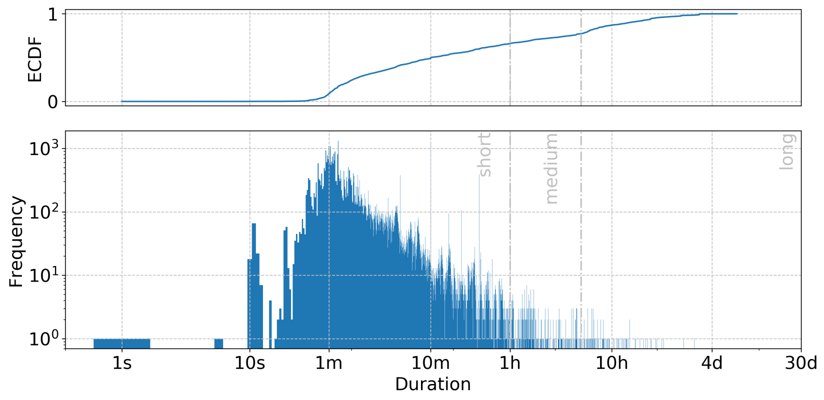


Figure 2.7: Distribution of job durations on the KIT system.

KIT and its workload dataset The Karlsruhe Institue of Technology ForHLR II System [21], shortened as KIT, is a HPC system located at the Karlsruhe Institue of Technology in Germany. The KIT system has two types of computing nodes: *thin* and *fat*. There are 1152 *thin* nodes with 20 cores and 64 GB memory each, and 21 *fat* nodes with 48 cores, 4 NVIDIA GeForce GTX980 Ti graphics cards, and 1 TB memory each.

The workload dataset contains 114,355 submitted jobs for one and a half years, from June 2016 to January 2018. Similar to the previous workloads datasets, as showed in Figure 2.7, short jobs prevail with 66.26% of the total

[20] `https://hpc.uni.lu/systems/gaia/`
[21] `https://www.scc.kit.edu/dienste/forhlr2.php`

number of jobs in the job duration distribution, followed by long jobs 22.55% and medium with 11.19%. The CPU time executed by all jobs in the workload rises to 562,772 hours, where long jobs represent the maximum resource usage with 92.08% of the total CPU time, 5.88% and 2.04% corresponds to medium and short jobs, respectively.

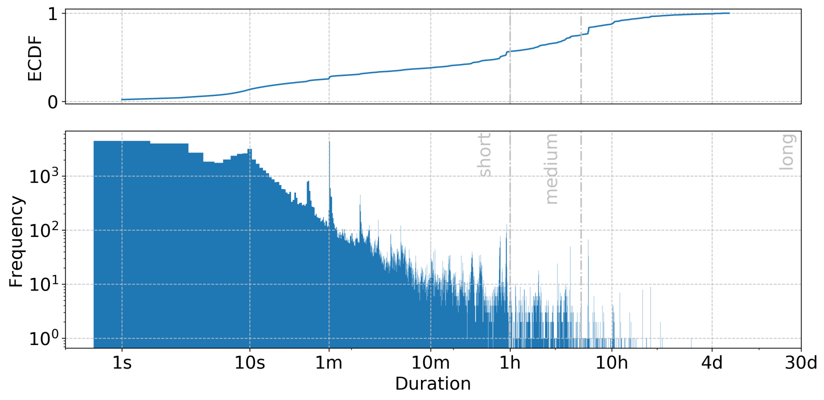


Figure 2.8: Distribution of job durations on the Seth system.

Seth and its workload dataset The Seth cluster[22] which was part of the High Performance Computing Center North of the Swedish National Infrastructure for Computing. Seth was composed of 120 nodes, each node contained two dual-core processors and 1 GB of RAM. The total system peak performance was 800 Gflops. Therefore, 480 cores and 120 GB of RAM in total. The workload dataset contains 202,871 jobs spanning through 4 years, from July 2002 to January 2006.

As shown in Figure 2.8, most of the half of jobs, specifically 57%, are short jobs, followed by long jobs representing 24%, and the remaining 19% belong to medium jobs. The total core hours are 953,912, and contributed mainly by long jobs, which represents the 87% of the total.

RICC and its workload dataset The RIKEN Integrated Cluster of Clusters, named RICC, is an HPC system located at RIKEN [23], an independent scientific research and technology institution of the Japanese government. The workload data corresponds to the massively parallel cluster, one of four of the clusters of RIKEN, which has 1024 nodes, each with 12 GB of memory and two 4-core CPUs, for a total of 12 TB RAM and 8,192 cores.

The workload dataset contains 447,794 submitted jobs from May 2010 to September 2010, so many jobs were submitted in a short period. This workload dataset also enforces the statement that short jobs prevail on HPC systems,

[22]`https://www.hpc2n.umu.se/resources/hardware/seth`
[23]`http://www.riken.jp/`

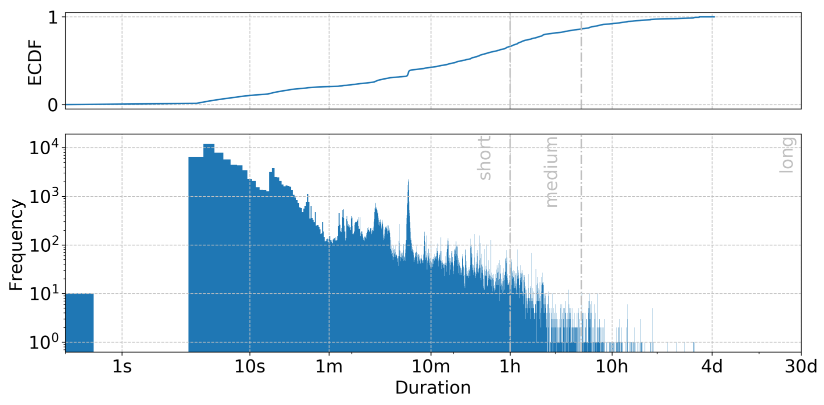


Figure 2.9: Distribution of job durations on the RICC system.

which is observable in Figure 2.9, 66.22% of the total number of jobs corresponds to this classification. Instead, 19.86% and 13.92% corresponds to medium and long jobs, respectively. The total CPU time is 1,473,652 hours, mainly executed by long jobs with a 83.20%, 12.64% and 4.16% corresponds to medium and short jobs, respectively.

MetaCentrum and its workload dataset The MetaCentrum system [24] is the national grid of the Czech republic. The workload dataset contains 5,731,100 jobs spanning through 2 years, from Jan 2013 to Apr 2015. During the recorded period, it was composed of 19 clusters with 495 nodes, 8412 cores and 10 TB of RAM in total. The current hardware of MetaCentrum is available online in its website. [25]

This workload dataset is, again, mainly composed of short jobs and follows a heavy-tailed distribution, see Figure 2.10: 79% of short jobs, 13% of medium jobs and 9% of long jobs. The total core hours reach to 18,444,716, where long jobs represents 89% of the total.

Workload datasets available in the Parallel Workload Archive In spite of Gaia, KIT, and MetaCentrum systems have GPU resources in their configuration, the workload does not contain information about the request of such a resource, therefore we cannot give these details either simulate them based on the available data. However, detailed information of the job request will be available in an actual system and will be provided by the WMS at dispatching time.

[24]`https://metavo.metacentrum.cz/en/index.html`
[25]https://metavo.metacentrum.cz/pbsmon2/hardware

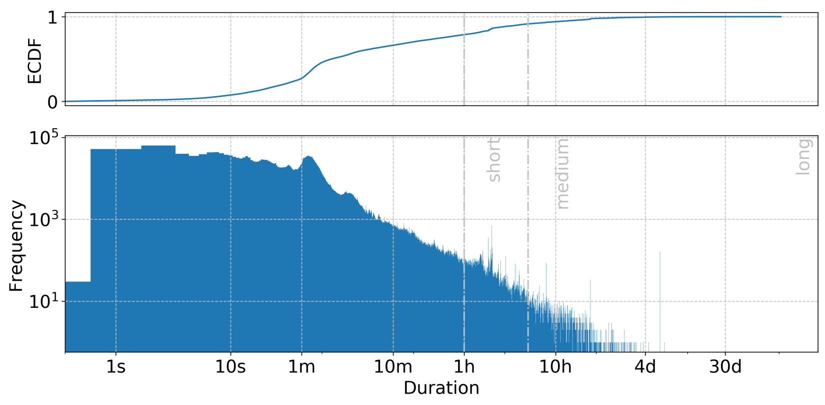


Figure 2.10: Distribution of job durations on the Metacentrum system.

2.2 Constraint Programming

This section is important to understand the nature of the dispatchers proposed in this dissertation. We briefly introduce Constraint Programming (CP) and present relevant concepts which will be used throughout the dissertation.

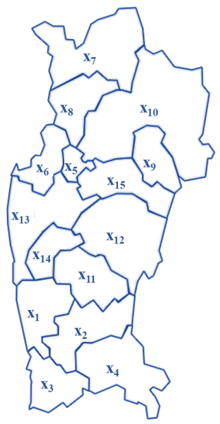
CP is a powerful paradigm for solving combinatorial search problems based on feasibility, that is finding feasible solutions, by focusing on the constraints and variables. CP bases its technology primarily on logic programming, graph theory, artificial intelligence, and operations research. To solve a problem using CP, it must be formulated as a Constraint Satisfaction Problem (CSP) by the process of modeling. A problem may have different alternatives for defining decision variables and constraints, which may affect the efficiency of the solution method. Therefore, an effective model for a given problem is a challenging task before to solve it. To find solutions, CP combines *search* and *constraint propagation* (See Section 2.2.2 and 2.2.3 respectively). In detail, a search algorithm searches in the space of possible assignments, while the *constraint propagation* keep all constraints consistent by reducing the domains of the variables, thus, reducing the effort during the search.

2.2.1 Constraint Satisfaction and Optimization Problems

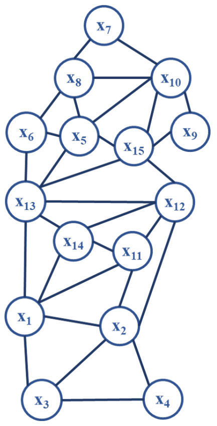
Constraint Satisfaction Problems (CSPs) are a standard, structured, and very simple representation of problems, where search algorithms can take advantage of the structure of the problem and use, in general, general-purpose heuristics to solve them. A CSP has a set of decision variables and a set of constraints. Each variable has a domain, which defines the possible values from a finite set that the variable can take, whereas each constraint restricts the possible values that the involved variables can at the same time take. Therefore, a solution to a CSP is an assignment of values to the variable such that all constraints are satisfied simultaneously.

Definition 2.2.1. CSP. A Constraint Satisfaction Problem (CSP) consists of a set of variables $X = \{x_1, \ldots, x_n\}$; a set of values, $D = \{a_1, \ldots, a_n\}$, where each variable $x_i \in X$ has an associated finite domain $dom(x_i) \subseteq D$ of possible values; and a collection of constraints $C_1, \ldots, C_m$. Each constraint C_i involves some subset of variables and specifies the possible combinations of values for that subset. A state of the problem is defined by an assignment of values to a subset of variables. An assignment on which all constraints are satisfied simultaneously is called a consistent assignment. A solution is found to a CSP when a complete assignment is performed, i.e. all variables have an assigned value. If no solution exists, the CSP is said to be inconsistent or unsatisfiable. Some CSPs require a solution that maximizes or minimizes an objective function.

We use the map-coloring problem as an example. In this problem, the aim is to color each region of a map in a way that neighbor regions are not colored with the same color. Consider the problem instance of coloring the map of *comumas* of the Región de Coquimbo, Chile; as depicted in Figure 2.11a. Following, we will define the corresponding CSP of this map. The *comunas* are represented by $X = \{x_1, \ldots, x_{15}\}$. We can consider 6 available colors based on the maximum number of neighbors of all *comunas*, max $|dom(X)|$, as to ensure a feasible solution. The domain for each x_i is $dom(x_i) = \{1, 2, 3, 4, 5, 6\}$, where each number represents a specific color. For instance, 1 is gray, 2 is blue, and so on.



(a) *Comunas* (administrative divisions) of Región de Coquimbo, Chile.



(b) Constraint graph of the map-coloring problem.

Figure 2.11: Representing the Región de Coquimbo map as a constraint graph.

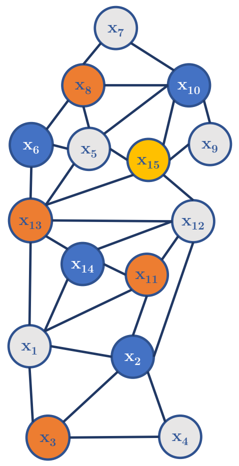
Regarding to the restrictions of the problem, i.e. neighboring regions require to have distinct colors, so we define an inequality constraint for each pair of neighbor *comunas*. We use a constraint graph, in this case Figure 2.11b, to visualize the CSP easily. The nodes correspond to variables of the problem and the arcs correspond to neighbor regions. Following, we consider x_1 as an example to define its constraints:

$$\begin{array}{ccc} x_1 \neq x_2 & x_1 \neq x_3 & x_1 \neq x_{11} \\ x_1 \neq x_{13} & x_1 \neq x_{14} & \end{array}$$

For the remaining *comunas*, $X - \{x_1\}$, we define the remaining inequalities as to complete all restrictions of the problem. Each constraint C_i is a relation (a set of tuples) over some set of variables, denoted by $vars(C)$. In general, the size of the set $vars(C)$ is called the arity of the constraint. In the previous case, we defined binary constraints, which is a constraint of arity two, but also there are the unary constraint which is a constraint of arity one, and a non-binary constraint which is a constraint of arity greater than two. Up to here the modeling process is finalized.

From now, solutions are obtained by using a Constraint Solver. A Constraint Solver finds assignments to all the variables that satisfies the problem constraints. Constraint Solvers are extended to involve, for example, finding optimal solutions according to one or more optimization criterion, finding all solutions, replacing (some or all) constraints with preferences, and considering a distributed setting where constraints are distributed among several agents. Constraint solvers search the solution space systematically or use forms of local search which may be incomplete (see Section 2.2.2). Search is interleaved with constraint propagation, which consists of propagating the information contained in one constraint to the neighboring constraints (see Section 2.2.3). Such inference reduces the parts of the search space that need to be visited.

Figure 2.12 depicts a solution of our example instance of the graph-coloring problem, which is an assignment of values to the variables such that all constraints are satisfied simultaneously:



$$\begin{array}{ccc} x_1 \leftarrow 1 & x_2 \leftarrow 2 & x_3 \leftarrow 3 \\ x_4 \leftarrow 1 & x_5 \leftarrow 1 & x_6 \leftarrow 2 \\ x_7 \leftarrow 1 & x_8 \leftarrow 3 & x_9 \leftarrow 6 \\ x_{10} \leftarrow 2 & x_{11} \leftarrow 3 & x_{12} \leftarrow 5 \\ x_{13} \leftarrow 3 & x_{14} \leftarrow 2 & x_{15} \leftarrow 4 \end{array}$$

Figure 2.12: A solution for the Región de Coquimbo instance

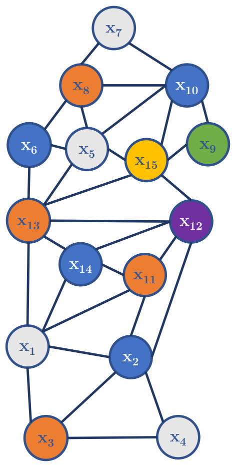
A CSP usually may contain multiple solutions, even when all the constraints are satisfied, however at this point there is no way to discriminate among them and such solutions appear equally good. To find the optimal solution to the CSP a function f must be either maximized or minimized depending on the requirements of the problem.

A Constraint Satisfaction Optimization Problem (CSOP) is defined as a CSP with an optimization function f which maps every solution to a numerical value. The task in a CSOP is to find the solution T such that the value of $f(T)$ is, without loss of generality, minimized. Therefore, in addition to problem constraints, an objective function f must be optimized. To do so, a variable c constrained to be equal to the objective function, $c = f(X)$, is included in the model. Then, a objective constraint, $c < f(T)$, is posted excluding solutions which are not better than the current T solution. The search continues until the entire search-tree is traversed. Therefore, the last solution found is proved optimal.

Therefore, getting back on track with the map-coloring problem, we can also find a solution using the minimum number of available colors, if the CSP is turned into an optimization problem. To do so, we consider an objective function defined as:

$$\min z = \max_{x_i \in X} x_i$$

Therefore, each variable $x_i \in X$ takes the minimum value of the colors for which all the constraint are satisfied. Figure 2.13 depicts the optimal solution for optimization version of the example with $z = 4$.



$x_1 \leftarrow 1$	$x_2 \leftarrow 2$	$x_3 \leftarrow 3$
$x_4 \leftarrow 1$	$x_5 \leftarrow 1$	$x_6 \leftarrow 2$
$x_7 \leftarrow 1$	$x_8 \leftarrow 3$	$x_9 \leftarrow 1$
$x_{10} \leftarrow 2$	$x_{11} \leftarrow 3$	$x_{12} \leftarrow 1$
$x_{13} \leftarrow 3$	$x_{14} \leftarrow 2$	$x_{15} \leftarrow 4$

Figure 2.13: Coloring the Región de Coquimbo map with the minimal number of available colors

2.2.2 Search

A systematic search algorithm, also known as a complete search algorithm, is the most simple way to find solutions to CSPs, and to guarantee (i) a solution if it exists, or (ii) to show that a CSP does not have a solution or to find an (iii) optimal solution. Conversely, a non-systematic search algorithm, also known as an incomplete search algorithm, is useful at finding a solution, without guarantee if one exists, in a large search space.

Backtracking search algorithms are the most relevant systematic algorithm in literature [130], and they have been studied different ways of improving their efficiency, including branching strategies, heuristics for variable and value ordering, randomization and restart strategies, among others.

The term Backtracking (BT) search is used for a depth-first search that chooses values for one variable at a time and backtracks when a variable has no values left to assign. This can be seen also as a depth-first traversal of a search tree, which is generated as the search progresses and represents alternative choices examined to find solutions. Extending a node in the search tree is called *branching*, and corresponds to select a variable and a value. Constraints are used to simplify the remaining subproblem and to check whether a node may possibly lead to a solution of the CSP and to prune subtrees containing no solutions. A node in the search tree is a dead-end if it does not lead to a solution.

So far, we assumed search works by posting an assignment, propagating and backtracking in case of failure to try another value. The previous scheme is called enumeration, or d-way branching, and tries all possible values. Another strategy is called *bisection* or *domain splitting* which posts two inequality constraints ($\leq$ and $>$) on each branch. Finally, the last branching strategy often used is the 2-way branching, also known as *binary choice points*, which corresponds to post an equality ($=$) and an not equality ($\neq$) constraints for the assignment. The last two strategies improves the search process since some values of the domain are deleted after posting the unary constraint which starts the propagation and detecting inconsistencies. All these strategies are identical if the domains are binary. Later, in Section 2.2.4, we describe in more detail a search strategy for scheduling problems which uses binary choice point strategy.

Branches are often ordered using heuristics, heuristic for variable and value selection. A variable ordering heuristic can be from one of two categories: domain size-based and CSP structure-based [130] heuristics. The first category of heuristics base their ordering on the current domain sizes of the unassigned variables [56, 61, 23, 13, 19]. Instead, CSP structure-based heuristics are based on the graph representation of a problem. These heuristics have some limitations which hinder they adoption like break down in the presence of global constraints, or because they are static or nearly static [8].

A value ordering heuristic choose the next value for a selected variable, and can be a simple heuristic such as picking the smallest value first and step-wise increase it to more complex, dynamic variable orderings evaluate information about the current node in the search tree to heuristically choose a promising value [39, 53, 54].

However, to identify which is the best strategy for a problem sometimes is a complex task, because among its instances different strategies may have different performances. Autonomous search [59] is a modern approach used to allow solvers to automatically re-configure their strategy to improve the search process when poor performances are detected via indicators gathered during the search. This is carried out using a hyperheuristic, which is a method to choose heuristics. This problem can be seen as an optimization problem, so as to improve the selection of heuristics simple metaheuristics are used [36, 37, 116, 117]. Autonomous Search is specially useful for users not having any expert knowledge for efficiently solving problems.

When a dynamic scheme for variable and value selection is used, *randomization* and *restart strategies* take a principal role during search. These techniques have been proposed to take advantage of a dynamic selection of variable or value ordering heuristic during the search, due to such heuristics perform a different number of mistakes, and during different stages during the search, then a large variability in performance between them. A periodically restarting with different variable orderings could eliminate the problem of "early mistakes" [57, 86].

2.2.3 Constraint Propagation

Constraint propagation, also known as constraint relaxation, filtering algorithms, narrowing algorithms, constraint inference, simplification algorithms, label inference, local consistency enforcing, rules iteration, chaotic iteration; is a very general concept which embeds any reasoning to explicitly forbid values or combinations of values for some variables of a problem because a given subset of its constraints cannot be satisfied otherwise [12].

Definition 2.2.2 (Constraint)**.** A constraint c is a relation defined on a sequence of variables $vars(c) = (x_{i_1}, \ldots, x_{i_{|vars(c)|}})$, where $vars(c)$ is called the scheme of c and $|vars(c)|$ is the arity of c. Constraint check is the process of testing if a tuple $\tau \in \mathbb{Z}^{|vars(c)|}$ satisfies c. The set of tuples $rel(c)$, called the constraint relation, specify the allowed variable-value combinations.

Therefore, the original problem is reduced to an equivalent but smaller problem, where a backtracking search commits into less inconsistent instantiations by detecting inconsistency earlier. Inconsistencies are detected by inference which can be global or local, however in practice, only local consistency is verified, i.e. individual constraints are examined because verifying global consistency is NP-Complete. Constraints of arity one are called *unary* constraints, those of arity two are *binary* and of a greater *arity* are non-binary. There are also constraints defined on an independent arity called *Global* constraints.

Different properties of consistency exists depending on the arity of the constraint. We describe here some of them, a more complete list can be found in [38, 12]:

- *node consistency*: A simple consistency property concerning unary constraints. We say a constraint c, with $|vars(c)| = 1$, is consistent if for all $a \in dom(x_1)$ satisfy c.
- *arc-consistency*: The most widely used local consistency for binary constraints. We say a constraint c, with $|vars(c)| = 2$, is arc-consistent for its variables if for every $a \in dom(x_1)$ there is some value $b \in dom(x_2)$ such that $rel(c)$ with $var(c) = (x_1, x_2)$ is satisfied.
- *generalized arc-consistency* (GAC): This consistency is not restricted to binary constraints, indeed it is a extension of arc-consistency with $|vars(c)| = n$. Then, for all variables $x \in var(c)$ and for all values $a \in dom(x)$ can be extended to all the other variables of the constraint in such a way the constraint is satisfied. GAC is one of the most commonly enforced forms of consistency in constraint programming.

In general, these properties are not sufficient conditions for obtaining solutions for CSPs, because more constraints are part of the same problem and the consistency is only checked locally. Therefore, constraint propagation is interleaved with search. Before searching, the problem is preprocessed to prune the domains of the uninstantiated variables, which at the beginning are all the variables of the problem. As the search continues, the domains of the uninstantiated variables shrink, as well as the size of the search tree, because of the local consistency must be maintained during all the search.

As the domain of a variable changes, each constraint which uses that variable is examined and the local consistency needs to be established if necessary. The local consistency is maintained through a systematic process called constraint propagation. We recall the constraint propagation is also due to variable assignments. The constraint propagation may result in the reduction of the domains of variables with filtering algorithms.

Global Constraints A global constraint is a constraint that captures a relation between a non-fixed number of variables. Usually, this kind of constraint can be expressed as the conjunction of several simpler constraints, but ease the modeling part is not one key aspect, they include specific filtering algorithms which capture better the structure of the problem. A complete list of global constraint can be found in the Global Constraint Catalog [26], which presents a list of 348 global constraints issued from the literature in CP and from popular constraint systems.

A well-known global constraint is the `alldifferent` constraint, which specifies that the values assigned to the variables must be pairwise different. As an example, we can consider the Sudoku problem, which consists of completing the assignment of a board using numbers from 1 to n^2 such that the entries in each row, each column, and each $n \times n$ sub-square are pairwise different. This problem can be easily modeled with $3n^2$ `alldifferent` constraints instead of using

[26] `https://web.imt-atlantique.fr/x-info/sdemasse/gccatold/`

pairwise differences for each region (rows, columns and sub-squares), which is more tedious and may not achieve a strong propagation. Many Sudoku instances can be solved using propagation [110, 115] showing the importance of this step.

We will introduce another well-known global constraints in Section 2.2.4 which are used in scheduling problems.

2.2.4 Scheduling in CP

Solving scheduling problems have been widely studied over the years by using CP, indeed such discipline is named Constraint-Based Scheduling, and is one of the most successful applications areas. This success is given by a virtuous circle originated by nature of CP, i.e. the integration of Artificial Intelligence and Operations Research. A detailed explanation of this synergy can be found in [6]. A general scheduling problem corresponds to allocate *scarce resources* to a given set of *activities* over time.

There exists different types of activities depending on the class of the problem: Non-preemptive scheduling, preemptive scheduling, and elastic scheduling. We focus on the first type of the scheduling problem, because it corresponds to the most similar class to model the job dispatching problem for HPC systems. A non-preemptive scheduling problem includes only activities which are started without interruption. An activity a_i can be easily encoded with three integer variables:

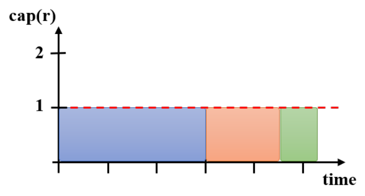
- s_i: Start time of the activity a_i.
- e_i: End time of the activity a_i.
- d_i: Duration of the activity a_i.

Since we are considering activities of a non-preemptive scheduling problem, these three variables must satisfy $s_i + d_i = e_i$. In scheduling problems, activities are usually restricted to starting and ending limits:

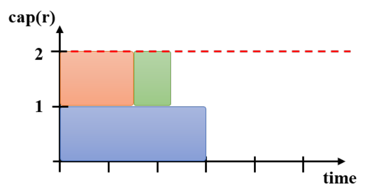
- Earliest Start Time (EST_i) of a_i: $min(dom(s_i))$
- Latest Start Time (LST_i) of a_i: $max(dom(s_i))$
- Earliest End Time (EET_i) of a_i: $min(dom(e_i))$
- Latest End Time (LET_i) of a_i: $max(dom(e_i))$

We can see that many constraints will emerge only for the definition of only one activity, for instance $s_i \in [EST_i, LST_i]$. Using constraint language, we introduce the concept *Conditional Interval Variable* (CIV) [76] to encapsulate all variables and constraints belonging to an activity. A CIV τ_i represents an activity a_i and defines the time interval during which a_i is executed. The properties $s(\tau_i)$ and $d(\tau_i)$ correspond respectively to the start time and the duration of the activity a_i. EST and LET are usually constrained by users to specify the release and deadline times of activities. CIVs allows to specify temporal relations either for unary (τ_i ends after t, τ_i ends at t, ...) and binary (τ_j ends after tau_i ends, τ_j ends after tau_i starts, ...) constraints. Another interesting feature is its execution status $x(\tau_i)$. An executed CIV, $x(\tau_i) = 1$, has a special meaning. Informally speaking, it corresponds to an interval variable which is concerned by all the constraints or expressions on variables on which it is involved in. Conversely, for a non-executed, $x(\tau_i) = 0$, it start, end and duration are meaningless.

Scheduling problems vary also depending on the type of scarce resources. These resources can be disjunctive and/or cumulative. In a disjunctive scheduling problem, all resources have capacity 1, i.e. only one activity can be executed at a time, as in Figure 2.14a. Whereas, in a cumulative scheduling problem, several activities can be executed in parallel without exceeding the resource capacity over time, as in Figure 2.14b. We represent the need for resources by activities over time through resource constraints. Given an activity a_i and a resource r, whose capacity is $cap(r)$, we represent the need of r by activity a_i as $req(a_i, r)$.



(a) A disjunctive resource.



(b) A cumulative resource.

Figure 2.14: Types of resources in scheduling problems

Disjunctive constraint In case of resources that can handle one activity at the time, the `disjunctive` constraint, $disjunctive(\tau)$, restricts the overlapping of two or more activities in time, i.e. $e(\tau_i) > s(\tau_j) \wedge s(\tau_i) < e(\tau_j)\ \forall i, j, i \neq j$. Such a situation is, of course, common in disjunctive scheduling but also occurs in cumulative scheduling, when the sum of the capacities required by two activities exceeds the capacity of the resource. This constraint has two main versions. The first one corresponds when the duration of the activities are fixed and the goal is to perform as many tasks as possible within their respective due-dates [7]. The second version deals with unbound duration, for which different propagation techniques have been proposed, such as constructive disjunction [64, 137], edge-finding [27, 132, 98], and the most popular, the time-tabling method [78].

Cumulative constraint The capacity constraints are enforced via the global `cumulative` constraint for all activities i ensuring that at each point in time the total $req(a_i, r)$ of the activities i using k that overlap at any point does not exceed cap_{r_k}. Formally,

$$\texttt{cumulative}(\tau, req_{A,r}, cap(r))$$

which holds iff $\sum_{i|s(\tau_i)\leq u<s(\tau_i)+d(\tau_i)} req_{a_i,r} \leq cap(r)$ for all u in time. Where τ is the set of CIVs, $req_{A,r}$ are the resource requirements and $cap(r)$ is the capacity of r.

Similar to the `disjunctive` constraint, the most commonly used propagation technique for the `cumulative` constraint is based on time-tables. This technique compute and maintain during the search the sum of every activity requirement $req_{A,r}$ at each time interval t. This information allows to restrict the domains of the start and end times of activities by removing the times that would necessarily lead to an over-consumption of the resource.

Alternative constraint The execution status of a CIV $(x(\tau_i))$ takes a special meaning with this constraint. Given a set of alternative possibilities $\{\tau_{i,1}, \ldots, \tau_{i,n}\}$ of a given τ_i, the `alternative` constraint ensures that if τ_i is executed, that is $x(\tau_i) = 1$, exactly m of the optional intervals are also executed and have the same duration, starting and ending times as τ_i. The signature of this constraint is:

$$\texttt{alternative}(\tau_i, \{\tau_{i,1}, \ldots, \tau_{i,n}\}, m)$$

which holds iff $\sum_j^n x(\tau_{i,j}) = mx(\tau_i)$ and $s(\tau_i, j) = s(\tau_i)$, $e(\tau_i, j) = e(\tau_i)$, $(\tau_i, j) = d(\tau_i)$ $\forall j \in n$.

Non-Overlapping Boxes Constraint The Non-Overlapping Boxes Constraint, also known as `diffn` constraint, was introduced in CHIP [11] to tackle multi-dimensional placement problems. In general, the `diffn` constraint holds if the set of objects defined in an n-dimensional space do not overlap, so there exists at least one dimension where two pairs of objects are disjunctive. For instance consider the example of four boxes which must be placed in a 6×7 space. The coordinates (x, y) and dimensions (dx, dy) of the boxes are:

box$_1$:	$x_1 \in [1,3]$	$dx_1 \in [2]$	$y_1 \in [1,3]$	$dy_1 \in [3]$
box$_2$:	$x_2 \in [1,3]$	$dx_2 \in [3]$	$y_2 \in [1,3]$	$dy_2 \in [2]$
box$_3$:	$x_3 \in [1,2]$	$dx_3 \in [1]$	$y_3 \in [1,4]$	$dy_3 \in [4]$
box$_4$:	$x_4 \in [1,6]$	$dx_4 \in [4]$	$y_4 \in [1,2]$	$dy_4 \in [1]$

Using the example, we redefine the data of the boxes in four sets, $X = [x_1, \ldots, x_4]$ and $Y = [y_1, \ldots, y_4]$, that correspond to the sets of the x and y coordinates, respectively; and $DX = [dx_1, \ldots, dx_4]$ and $DX = [dx_1, \ldots, dx_4]$, to the sets of the x and y dimensions, respectively. Finally, we set the maximum values of the space as $X_{max} = 6$ and $Y_{max} = 7$.

To restrict the overlapping we post a $diffn$ constraint as follows:

$$\texttt{diffn}((X, DX, Y, DY, X_{max}, Y_{max})$$

Figure 2.15 shows all solutions of the example. [27]

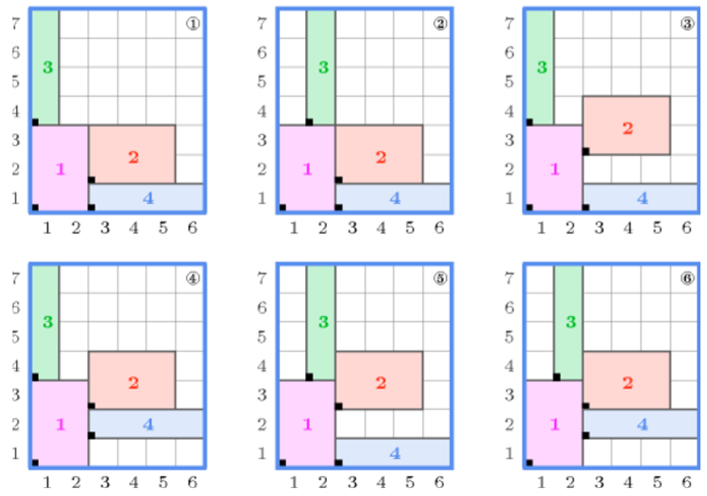


Figure 2.15: Possible solutions for the `diffn` example.

Search in scheduling Search strategies presented in Section 2.2.2 do not utilize any information specific to scheduling problems, tending to show a very slow performance. Therefore, specific search strategies for scheduling problems has been proposed in literature [97, 107, 55, 84]. The Set Start Times (SST) is a widely-used search strategy solving scheduling problems also known as Schedule Or Postpone, which is based on Priority Rules-Based scheduling, thus finds good solutions early, and implicitly makes ordering decisions. The algorithm of the SST is presented in Algorithm 1.

Algorithm 1: The Set Start Times algorithm

```
1 solution or failure ← SST (assignment, CSP)
2 begin
3     var ← SELECT_INTERVAL(assignment, CSP)
4     if var <> NULL then return solution or failure
5     return SST(ScheduleOrPostpone(var))
6 end
```

At each node of the search-tree this strategy selects the interval τ_i with the minimal EST and minimal LET to break ties. Create a choice point (ScheduleOrPostpone) to allow backtracking: on the left branch $s(\tau_i) = EST_i$, and on the right branch τ_i is marked as non-selectable until its earliest start time is modified by propagation, i.e. postponed.

[27]Source: `https://sofdem.github.io/gccat/gccat/Cdiffn.html`

2.3 CP-based online job dispatchers for HPC systems

We already introduced Constraint Programming (CP), however we did not mention why using it to solve the job dispatching problem in HPC system may be an interesting option. CP is a declarative programming paradigm for modeling and solving constraint satisfaction and optimization problems [105]. While it has its roots in artificial intelligence, the last decades have witnessed its successful cross-fertilization with related disciplines such as operations research, metaheuristics, SAT, and more recently, machine learning. [89, 131] provide some perspectives on what has been achieved in the last twenty years of CP research in tackling constraint optimization problems. CP has been widely used for scheduling problems, in various contexts such as transportation [107, 104], staff scheduling [135], sports competition scheduling [100], etc.

Since the job dispatching problem can be naturally framed as resource allocation and scheduling problem, CP-based job dispatchers have been proposed given that the promising application of CP in the scheduling and allocation area [6]. All this success is due to specific types of decision variables, global constraints, optimization functions and search algorithms dedicated to the dispatching domain. There are two main state-of-the-art CP-based dispatchers for HPC systems available in the literature. The first CP-based dispatcher [9], the entire dispatching problem is modeled and solved using a CP solver. The second dispatcher [18] instead relies on a hybrid method. While the scheduling problem is modeled and solved in a CP solver, the allocation problem is solved separately using a heuristic search algorithm. We will refer to them as `PCP` and `HCP`, respectively, to mean the use of a Pure CP and a Hybrid CP method in their dispatching algorithms.

Scheduling In both `PCP` and `HCP`, the scheduling problem is modeled with interval variables [75]. An interval variable $\tau_i \in \tau$ represents a job i and defines the time interval during which i runs. At a certain dispatching time t, there may already be jobs in execution which were previously scheduled and allocated. We refer to such jobs as running jobs. The scheduling model considers in the τ variables both the running jobs and the queued jobs in Q. The properties $s(\tau_i)$ and $d(\tau_i)$ correspond respectively to the start time and the duration of the job i. Since the actual runtime duration d_i^r of a running or queued job i is unknown at the modeling time, `PCP` and `HCP` rely on an estimation and use the expected duration d_i for $d(\tau_i)$. Thus we have $d(\tau_i) = d_i$ for the queued jobs and $d(\tau_i) = s(\tau_i) + d_i - t$ for the running jobs. While the start time of the running jobs have already been decided, the queued jobs have $s(\tau_i) \in [t, eoh]$, where eoh is the end of the worst-case makespan calculated as $t + \sum_{\tau_i} d(\tau_i)$. Expected durations d_i are supplied by the users. In the absence of this information, the dispatchers use the default wall-time of the queue. It is important to note that even user-supplied values tend to be equal to the wall-time of the queue, which is indeed the maximum allowed value for d_i. We will refer to the use of such d_i

to define $d(\tau_i)$ as the *wall-time approach.*

Unlike `PCP`, `HCP` searches for a start time for the first m jobs in Q (referred to as $\bar{Q}$). The remaining jobs in $Q \setminus \bar{Q}$ are still in the model, but they are postponed to the end of the makespan by fixing their start time as $s(\tau_i) = eoh - d(\tau_i)$.

The capacity constraints in `PCP` are enforced via a `cumulative` constraint as

$$\forall n \in N \; \forall r \in R \quad \texttt{cumulative}(\tau, req_r, cap_{n,r})$$

which holds iff $\sum_{i|s(\tau_i) \leq u < s(\tau_i)+d(\tau_i)} req_{i,r} \leq cap_{n,r}$ for all u in the makespan. Thus, for all $n \in N$ and for all $r \in R$, it is ensured that at any given time in the makespan the total $req_{i,r}$ of the jobs i using r does not exceed $cap_{n,r}$. In `HCP`, resources of the same type across all nodes are considered as a pool of resources, hence the `cumulative` constraints are posted for each $r \in R$ with the total capacity $Cap_r^T = \sum_{n \in N} cap_{n,r}$. Any infeasibility that may be introduced due to this modelling choice is fixed during the allocation phase. Moreover, `HCP` considers also power as a resource type r with its own total capacity Cap_{power}^T, which is a user-defined power cap, and the `cumulative` constraint ensure that the total power consumed by the jobs cannot exceed Cap_{power}^T.

Both dispatchers consider the objective function which minimizes the sum of the waiting times of the jobs. In `PCP`, the objective function is formalized as:

$$\sum_{\tau_i} \max(0, \frac{s(\tau_i) - q_i - ewt_i}{ewt_i})$$

which is a weighted sum so as to give priority to the jobs that stay in the queue longer than their ewt_i. The ewt_i value is the average waiting time of the queue where i is submitted, and is obtained by analyzing the execution traces of the workload dataset. Originally, these values where obtained by analyzing the Eurora workload dataset which was collected by the PBS dispatcher [63].

In the objective function of `HCP`, the weights are slightly different, giving priority to the jobs of the queues with lower expected waiting times:

$$\sum_{\tau_i} \frac{\max(ewt_i)}{ewt_i} * (s(\tau_i) - q_i)$$

We will explain later how the corresponding scheduling models are solved by `PCP` and `HCP`.

Allocation In `PCP`, the total amount of requested resources $req_{i,r}$ of a job i is divided into rn_i identical jobs units $u_{i,j}$, where rn_i is the maximum number of requested nodes for i and is supplied by the user during job submission. Each job unit $u_{i,j}$ requires $req_{i,r}/rn_i$ amount of resources for each $r \in R$. The units of a job can be allocated on the same or different nodes, depending on the availability in the system. The number of $u_{i,j}$ which can be allocated on a node n is calculated as $p_{i,n} = min(rn_i, \min_{r \in R} \lfloor \frac{cap_{n,r}}{req_{i,r}/rn_i} \rfloor)$. We denote with $[u_{i,j,n}]$ the sequence of $p_{i,n}$ possible allocations of the jobs units $u_{i,j}$ of i on n, where each element in the sequence is a Conditional Interval Variables (CIVs)

[75]. So, the allocation problem is modelled using CIV, which has an interesting property, if a CIV is set as non-executed, it is not considered by any constraint or expression on interval variables it is involved in. Conversely, interval variables used in the scheduling model are always executed, thus, are always considered and have always a starting time.

Thus, $u_{i,j,n}$ represents the utilization of the $req_{i,r}/rn_i$ amount of resources of a node n during $[s(\tau_i), s(\tau_i) + d(\tau_i)]$. The fact that rn_i allocations among $[u_{i,j,n}]$ should be chosen is enforced as:

$$\forall \tau_i \quad \texttt{alternative}(\tau_i, [u_{i,j,n}], rn_i)$$

which ensures rn_i CIVs in $[u_{i,j,n}]$ to have $s(u_{i,j,n}) = s(\tau_i)$ and $d(u_{i,j,n}) = d(\tau_i)$ i.e. $u_{i,j,n} \equiv \tau_i$, if $s(\tau_i)$ is assigned a value.

Instead in `HCP`, it is solved by a PRB algorithm for the jobs which have $s(\tau_i) = t$ after the scheduling model is solved. This heuristic algorithm iteratively tries to allocate each scheduled job using the best-fit allocation strategy. The jobs are chosen based on their priority. The jobs that have been waiting the longest at time t have the highest priority. Such a priority is calculated in line with the priority of the jobs in the objective function:

$$\frac{\max(ewt_i)}{ewt_i} * (t - q_i)$$

.

As a tie breaker, job demand is used, which is the job's resource requirements multiplied by job duration $d(\tau_i)$. Hence, among the high priority jobs, those that have requested fewer resources and have shorter durations have further priority. Since the scheduling decision may contain some inconsistencies due to considering the resources of the same type as a pool, a job may not be allocated, in which case it is postponed to the next dispatching time.

Search To solve the scheduling and the allocation model altogether, `PCP` uses the self-adapting large neighborhood search algorithm [74] which is the default search available in the solver where `PCP` is implemented [77]. `HCP` instead uses a custom search algorithm derived from the schedule-or-postpone algorithm [96] to solve the scheduling model. The criteria used to select a job among all the available ones at each decision node follows the priority rule used in the PRB allocation algorithm, thus preferring the jobs that can start first and whose priority are highest. Note that the priorities are calculated once statically at the dispatching time t before search starts. Due to problem complexity, search in both `PCP` and `HCP` is bounded by a time limit δ. Thus, the best solution found within the limit is the dispatching / scheduling decision. If, however, no solution is found within the limit, the search is restarted with an increased time limit $2 * \delta$. This procedure continues while no solution is found and $\delta \geq \delta_{max}$, where δ_{max} is the maximum time available to generate a decision.

Scalability of the models For a given instance of the on-line job dispatching problem, `PCP` will use $|Q|$ decision variables to model the scheduling problem and $\sum_{i\in\bar{Q}}\sum_{n\in N} p_{i,n}$ decision variables to model the allocation problem, where $p_{i,n} = min(rn_i, \min_{r\in R}\lfloor\frac{cap_{n,r}}{req_{i,r}/rn_i}\rfloor)$, and, rn_i and $req_{i,r}$ are the number of requested nodes and the number of requested r resources by i, respectively; $cap_{n,r}$ is the capacity of the resource r in n. To illustrate the number of decision variables employed in the PCP model, let us consider a simple serial job i, i.e. $rn_i = 1$, so the decision variables to model i will be $1 + |N|$, with N as the set of computing nodes. If the system is small, such as Eurora which has 64 nodes, with half of them corresponds to nodes with GPU and the other half MIC nodes, the model will use 65 decision variables in the worst case, instead, if the job requests MIC or GPU resources, it will be 33 decision variables. Thus, this model in a bigger system will use more decision variables, even more, if jobs are highly parallel ($rn_i > 2$) and still more if many jobs are in $|Q|$. Thus, this model may not be suitable on a system with a higher number of nodes due to the increment of decision variables and consequently requiring more time to generate a dispatching decision.

Conversely, HCP presents a simpler model with only $|Q|$ decision variables, being suitable for any-size of HPC systems. However, since the allocation problem is uncoupled, the generated solution is loosely generated in relation to the actual availability of the system, which may produce low-quality solutions.

Chapter 3

Accasim: A Workload Management System simulator

HPC systems have become fundamental tools to solve complex, compute-intensive, and data-intensive problems in different research and business fields. As the demand for HPC technology continues to grow, a typical HPC system receives a large number of variable requests by its end users. These precedents call for the efficient management of the submitted workload and system resources. To study the impact of different configurations in such management, in this chapter, we introduce a workload management simulator named AccaSim.

One of the challenges of job dispatching research is the intensive experimentation necessary for evaluating and comparing various dispatchers in a controlled environment. The experiments differ under a range of conditions with respect to the workload, the number and the heterogeneity of resources, and the dispatching algorithms. Using a real HPC system for experiments is not realistic for the following reasons. First, researchers may not have access to a real system. Second, it is impossible to modify the hardware components of a system, and often unlikely to access its Workload Management System (WMS) for any type of alterations. And finally, even with a real system permitting modifications in its WMS, it is inconceivable to ensure that distinct dispatchers process the same workload, which hinders fair comparison. Therefore, simulating a WMS is essential for conducting controlled dispatching experiments. However, simulators present also limitations because they fail to capture all the dynamic, variety, and complexity of real-life conditions. Instead, simulators are dedicated to cover one specific aspect, which in case of AccaSim is the study of scheduling and allocation (dispatching) methods. In spite of this, AccaSim allows users to extend the basic behavior focused on dispatching research, as to include distinct aspects and cover more conditions of real systems.

In the following sections, we give details of Accasim, which is characterized

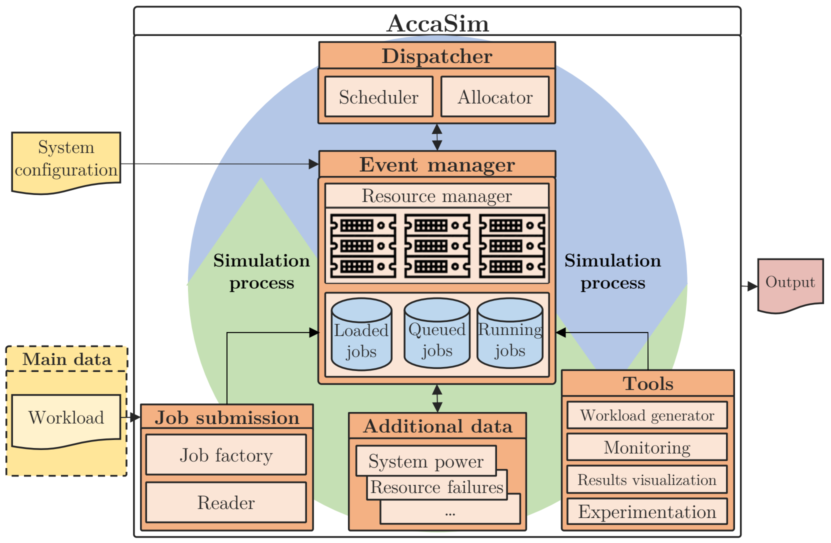


Figure 3.1: AccaSim architecture.

by being a simulator scalable to large workload datasets, with easy customization allowing users to carry out experiments across different workload sources, resource types, and dispatching algorithms. In addition, it allows to include custom behaviors, such as power and energy consumption and failures of resources. AccaSim allows users to easily represent various real HPC systems, develop novel advanced dispatchers, and evaluate them in a convenient way across different workload sources.

This chapter belongs to [48], a conference publication, and [49], an extended version published in the Cluster Computing Journal.

3.1 Architecture and Main Features

AccaSim enables to simulate the WMS of any real HPC system with minimum effort and facilitates the study of various issues related to dispatchers, such as feasibility, behavior, and performance, accelerating the dispatching research process. In this section, we present the architecture and highlight the main features of AccaSim.

AccaSim is designed as a discrete event simulator. The simulation is guided by certain events that belong to a real HPC system. These events are mainly collected from the workload and correspond to the job submission, starting and completion times, referred to as T_{sb}, T_{st} and T_c, resp. The architecture of AccaSim is depicted in Figure 3.1. Since there are no real users for submitting jobs nor real resources for computation during simulation, the first step for starting a simulation is to define the synthetic system with its jobs and resources.

Job submission. This component mimics the job submission of users. The main input data is the workload dataset provided in the form of a file which includes job descriptions. The default *reader* subcomponent reads the input file in Standard Workload Format (SWF)[46] and passes the parsed data to the *job factory* subcomponent for creating the synthetic jobs for simulation, keeping the information related to their identification, submission time, duration and request of system resources. The *job factory* can extend this basic information with additional attributes for the synthetic jobs, such as job duration estimation which is a useful information for many dispatching algorithms [51]. The synthetic jobs are then mapped to the *event manager* component, simulating the job submission process. The main data input is customizable in the sense that any workload dataset file can be used. This is possible thanks to the *reader* which can be easily adapted to parse any workload dataset file format. Consequently, AccaSim can be employed with any workload source corresponding to an existing workload dataset or to a synthetic one produced by a workload generator.

Event manager. This is the core component of the simulator, which mimics the behavior of the synthetic jobs and the presence of the synthetic resources, and manages the coordination between the two. Differently from a real WMS, the *event manager* tracks the jobs during their artificial life-cycle by maintaining all their possible states "loaded", "queued", "running" and "completed" via certain events. During simulation, at each time point t:

- the *event manager* checks if $t = T_{sb}$ for some jobs. If the submission time of a job is not yet reached, the *event manager* assigns the job the "loaded" state meaning in the real context that the job has not yet been submitted. If instead the submission time of a job is reached, the *event manager* updates its status to "queued";
- the *dispatcher* component gives a dispatching decision on (the subset of) the queued jobs, assigning them an immediate starting time. The *event manager* reveals that $t = T_{st}$ for some waiting jobs and consequently updates their status to "running";
- the *event manager* checks if $t = T_c$ for currently running jobs. Since these jobs were dispatched in a previous time point, their starting and completion times are known. The completion time of a job is the sum of its starting time and duration, which are known from the workload data. If the completion time of a job is reached, the *event manager* updates its status to "completed".

The *resource manager* subcomponent of the *event manager* defines the synthetic resources of the system using a system configuration file as input, and then mimics their allocation and release at the job starting and completion times. Hence, at a time point t, if a job starts, the *resource manager* allocates for the job the resources decided by the *dispatcher*; and if it completes, the *resource* *manager* releases its resources. The system configuration file can be customized according to the needed types of resources, which renders the simulation of a system possessing heterogeneous resources possible.

AccaSim is designed to maintain a low consumption of memory for scalability to large workload datasets, therefore job loading is performed in an incremental way, loading only the jobs that are near to be submitted at the corresponding simulation time, as opposed to loading them once and for all. Moreover, completed jobs are removed from the system so as to release space in the memory.

Dispatcher. This component, responsible for generating a dispatching decision, interacts with the *event manager* for retrieving the current system status regarding the queued jobs, the running jobs, and the availability of the resources. Note that the *dispatcher* is not aware of job durations. This information is known only by the *event manager* to stop the jobs at their completion time in a simulated environment. Therefore, the dispatching decision can be solely based on job duration estimations which are supplied as a job attribute. This has no impact on the execution of jobs, which are always allowed to run for their entire duration, despite the presence of estimation errors. The *scheduler* and the *allocator* subcomponents of the *dispatcher* are customizable according to the algorithms of interest. Currently implemented and available schedulers are: First In First Out (FIFO), Shortest Job First (SJF), Longest Job First (LJF) and Easy Backfilling with FIFO priority (EBF) [136]; and allocators are: First-Fit (FF) which allocates to the first available resource, and Best-Fit (BF) which sorts the resources by their current load (busy resources are preferred first), thus trying to fit as many jobs as possible on the same resource, to decrease the fragmentation of the system.

Additional data. It has been shown in the last decade that system performance can be enhanced greatly if the dispatchers are aware of additional information regarding the current system status, such as energy and power consumption of the resources [138, 4, 15, 18], resource failures [81, 20], and the heating/cooling conditions [127, 5]. The *additional data* component of AccaSim provides an interface to integrate such extra data to the system which can then be utilized to develop and experiment with advanced dispatchers which are for instance energy and power-aware, fault-resilient and thermal-aware. The interface lets receive the necessary data externally from the user, make the necessary calculations together with some input from the *event manager*, all customizable according to the need, and pass back the result to the *event manager* so as to transfer it to the *dispatcher*.

Output. The output file contains two types of data. The first regards the execution of the dispatching decision for each job, such as the starting time, the completion time and its resource allocation, which gets updated each time a job completes its execution. This type of data can be utilized to contrast the quality of the dispatching decisions from different perspectives. An example is the effect on synthetic system resource utilization: how many and which resources are used in the system, and how they are distributed over the nodes. Another example is the impact on system performance. With the increasing trend in employing HPCs for real-time applications which cannot tolerate delays [90], some critical aspects of system performance are job response times and system throughput. The second type of output data regards the simulation process, specifically the CPU time required by the simulation tasks like job loading, generation of the dispatching decision, and the total amount of memory used during simulation, which gets updated at each simulation time point. This type of data can be used, for instance, to evaluate the performance of the simulator, as well as the performance of the dispatchers in terms of the time they incur for generating a decision.

Tools. The *tools* let users follow the simulation process and facilitate their dispatching experimentation. We will demonstrate their utility in Section 3.5. The *monitoring* subcomponent includes the *system status* and *system utilization* subcomponents. The *system status* allows tracking the current system status, such as the number of queued jobs, the running jobs, the completed jobs, the availability of the resources, etc. The *system utilization* instead shows in a GUI a representation of the allocation of resources by the running jobs during the simulation.

The *results visualization* subcomponent renders the automatic generation of different types of plots for evaluating the quality of dispatching decisions as well as the performance of the dispatchers. The *experimentation* subcomponent instead renders the automation of complex experiments. After configuring the simulator with a workload dataset, a system to simulate, and a set of dispatchers, the *experimentation* performs a simulation for each dispatcher and then produces comparative plots through the *results visualization*.

When doing dispatching research with a real workload dataset, users could face issues such as the dependency on the real system configuration which hinders testing with other system configurations, the small size of the dataset preventing scalability tests, or the unavailability of certain data in the dataset for testing specific cases. To tackle this, AccaSim provides a *workload generator* subcomponent which produces a synthetic workload dataset. This subcomponent exploits the data contained in a real workload dataset by mimicking, through statistical methods, its distributions for job submission times, jobs resource requests, and job durations. The generated dataset is written to a file in the SWF format. Other file formats can as well be considered by customizing its subcomponents.

To highlight the *main features*, (i) AccaSim is designed to be scalable to large workload datasets; (ii) AccaSim is customizable in its workload source, resource types, and dispatching algorithms, providing maximum flexibility in representing a WMS; (iii) AccaSim enables users to develop novel advanced dispatchers by exploiting information regarding the current system status, which can be extended for including custom behaviors such as energy and power con-

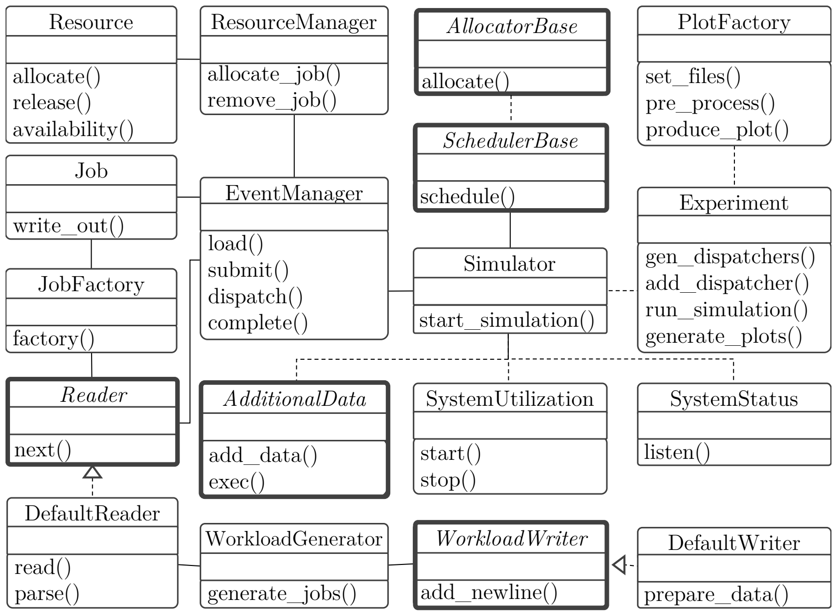


Figure 3.2: AccaSim class diagram.

sumption and failures of the resources; (iv) Accasim provides output data and automated tools to analyze the results, to follow the simulation process and facilitate dispatching experimentation.

3.2 Implementation, Customization, and Instantiation

In this section, we briefly describe AccaSim's implementation and customization, and show its various instantiations. This not only serves to depict the internal organization of AccaSim, but also provides evidence on how easy it is to use and customize.

AccaSim is implemented in Python which is an interpreted, object-oriented, high-level programming language, freely available for any major operating system, and is widely used in academia and industry.[28] All the dependencies used by AccaSim are part of Python 3.5 and newer versions, except the matplotlib, scipy, sortedcontainters and psutil packages which can be easily installed using the pip management tool. The source code is available under MIT License. User and API documentations can be found on the AccaSim website.[29] A release version is available as a package in the PyPi repository.[30] Customization

[28]`https://www.python.org/events/python-events/`
[29]`http://accasim.readthedocs.io/en/latest/`
[30]`https://pypi.org`

```
 1 from accasim.base.simulator_class import Simulator
 2 from accasim.base.scheduler_class import FirstInFirstOut
 3 from accasim.base.allocator_class import FirstFit
 4 from accasim.utils.plot_factory import PlotFactory
 5
 6 workload = 'workload.swf'
 7 sys_cfg = 'sys_config.json'
 8
 9 allocator = FirstFit()
10 dispatcher = FirstInFirstOut(allocator)
11 simulator = Simulator(workload, sys_cfg, dispatcher)
12 output_file = simulator.start_simulation()
13
14 plot_factory = PlotFactory('decision', sys_cfg)
15 plot_factory.set_files(output_file, 'my_plot')
16 plot_factory.produce_plot('slowdown')
```

Figure 3.3: A basic AccaSim instantiation.

is driven by the abstract classes and the inheritance capabilities of Python. The UML class diagram of the main classes is shown in Figure 3.2 where the abstract classes associated to the customizable components are highlighted in bold.

The simulator. A basic AccaSim instantiation is detailed in Figure 3.3. A simulator object is created in line 11 by instantiating the *Simulator* class. It receives as arguments a workload dataset file in, for instance, SWF, a system configuration file in JSON format, and a dispatcher object, with which the synthetic system is generated and loaded with all the default features.

The workload dataset file is handled by an implementation of the abstract *Reader* class, which is the SWF-based *DefaultReader* by default. The file is read and parsed by the *read()* and *parse()* methods. By implementing the *Reader* class appropriately, AccaSim can be customized to read any workload dataset file format beyond SWF, or to read workloads from any source, not necessarily from a file. The system configuration file, which is processed by the *ResourceManager* class, defines the synthetic resources. The file has two main contents. The first specifies the resource types and their quantity in a node belonging to a group, which is useful for modeling HPC systems possessing heterogeneous resources. The second, instead, defines the number of nodes of each group. See Figure 3.6 for an example. The user is free to mimic any real system by customizing this configuration file suitably.

The dispatcher object is composed by implementations of the abstract *SchedulerBase* and *AllocatorBase* classes. Both classes must implement their main methods, *schedule()* and *allocate()* respectively, to deal with the scheduling and the allocation decisions of the dispatching. This illustrative instantiation exemplifies a specific instance of the *Simulator* class, using as scheduler the *FirstInFirstOut* class, which implements *SchedulerBase* with FIFO, and as allocator the *FirstFit* class, which implements *AllocatorBase* using FF. Both the *FirstInFirstOut* and *FirstFit* classes are available in the library for importing, as done in lines 2-3 of Figure 3.3. AccaSim can be customized in its dispatch-

```
 5 [...]
 6 from accasim.base.scheduler_class import ShortestJobFirst
 7 from accasim.experimentation.experiment import Experiment
 8
 9 experiment = Experiment('my_experiment', workload, sys_cfg)
10 sched_list = [FirstInFirstOut, ShortestJobFirst]
11 alloc_list = [FirstFit]
12 experiment.gen_dispatchers(sched_list, alloc_list)
13 experiment.run_simulation()
```

Figure 3.4: An AccaSim instantiation using the experimentation tool.

ing algorithm by implementing the abstract *SchedulerBase* and *AllocatorBase* classes as desired.

In line 12, the *start_simulation()* method launches the simulation with the following optional arguments:

```
simulator.start_simulation(
                    system_status=True,
                    system_utilization=True,
                    additional_data=None)
```

which serve to require the use of the *system status*, the *system utilization*, and the *additional data* tools of the simulator. The additional_data argument is an array of objects where each object is an implementation of the abstract *AdditionalData* class, giving the possibility to customization in terms of the extra data that the user may want to provide to the system for dispatching purposes. After the simulation is finished, the output data file is returned.

The last three lines in Figure 3.3 serve to use the automated plot generation tool. In line 14, the *PlotFactory* class is instantiated using two arguments. The first indicates the plot type to be produced, as a decision-related or performance-related type. A decision-related plot shows metrics related to the quality of the dispatching decision, such as the job slowdown [43] or queue size, while a performance-related plot serves to show metrics related to the performance of the dispatcher, such as the average CPU time at a simulation time point. Examples of such plots will be shown in Section 3.5. The second argument is instead the system configuration file which is necessary for the resource specific plots. In line 15, the output file of the simulator is set to be analyzed through the *set_files()* method, together with a label to be used in the plots. Finally, the *produce_plot()* method produces the desired plot as specified in its argument.

The experimentation tool. In Figure 3.4, an AccaSim instantiation that uses the *experimentation* tool is detailed. The first 4 lines related to imports and assignment statements are the same as lines 2, 3, 6 and 7 in Figure 3.3 and are therefore omitted. An experiment object is created in line 9 by instantiating the *Experiment* class which takes as arguments the name of the experiment (which is used to name the output directory as well), the workload dataset file, and the system configuration file, along with the the optional arguments supported by the *Simulator* class. In line 12, the dispatchers of interest are

```
1 from accasim.experimentation.workload_generator import WorkloadGenerator
2
3 workload = 'real_workload.swf'
4 sys_cfg = 'sys_config.json'
5 performance = {'core': 1.667}
6 request_limits = {'min': {'core': 1, 'mem': 256}, 'max': {'core': 8, 'mem': 1024}}
7
8 gen = WorkloadGenerator(workload, sys_cfg, performance, request_limits)
9 jobs = gen.generate_jobs(500000, 'new_workload.swf')
```

Figure 3.5: A basic workload generator instantiation.

generated through the *gen_dispatchers()* method, which accepts as arguments a list of scheduler and allocator classes. In this illustrative instantiation of the *Experiment* class, we use the *FirstInFirstOut* and the *ShortestJobFirst* classes which implement FIFO and SJF scheduling, as well as the *FirstFit* class which implements the FF allocation. All these classes are available in the library for importing, as done in lines 6-7 of Figure 3.4. The *gen_dispatchers()* method then automatically creates the dispatchers corresponding to all possible combinations between the schedulers and the allocators, facilitating greatly the conduction of experiments on large sets of dispatchers. If users wish to experiment with a specific dispatcher, it can be formed by instantiating the corresponding implementation of *SchedulerBase* and then passing the object to the *add_dispatcher()* method, similarly to what we have shown in the lines 9-11 in Figure 3.3 when instantiating the *Simulator* class. Finally in line 13, the experiment is launched with the *run_simulation()* method which performs simulations for all configured dispatchers and produces all the available plots.

The workload generator tool. The workload dataset file can refer to a real workload dataset extracted from an HPC system, or to a synthetic one generated through an external workload generator such as AccaSim's own *workload generator* tool. Figure 3.5 shows its basic instantiation. A generator object is created in line 8 via the *WorkloadGenerator* class which is available in the library for importing, as done in line 1. It receives as arguments a real workload dataset file to be mimicked, a system configuration file, and variables regarding performance and request limits. The performance variable is a dictionary storing the performance of each processing unit as a unit-value pair. The request_limits variable instead defines the minimum and maximum request of each resource type available in the system. Finally, the jobs are generated in line 9 using the *generate_jobs()* method, which receives as arguments the number of jobs and the name of the output file in which the generated workload dataset is saved.

As in the case of the simulator, the input workload dataset file is parsed by an implementation of the abstract *Reader* class, which is *DefaultReader* and implements an SWF reader by default. The output file is instead written through an implementation of the abstract *WorkloadWriter* class, which is the SWF-based *DefaultWriter* by default. Similar to the *Reader*, the output file format can be customized by implementing the *WorkloadWriter* suitably. It is also possible to customize the job generation process via the optional arguments of the *WorkloadGenerator* constructor, as detailed in the AccaSim documentation.

3.3 Related Work

HPC systems have been simulated from distinct perspectives, for instance to model their network topologies [1, 69, 91] or storage systems [114, 95]. There also exist simulators dealing with the duties of a WMS, as in our work, which are mainly focused on job submission, resource management and job dispatching.

To the best of our knowledge, the WMS simulators most similar to AccaSim are ScSF, Batsim, and Alea. The ScSF simulator [103] emulates a real WMS, Slurm Workload Manager[31], which is popular in many HPC systems. In [87, 122] Slurm is modified to provide synthetic job submission, resource management and job dispatching through distinct daemons which run in diverse virtual machines and which communicate over RPC calls, and a dedicated simulator is implemented. ScSF extends this simulator with automatic generation of synthetic job descriptions based on statistical data, but does not give the possibility to read real workload datasets. The dependency on a specific WMS complicates the customization, and together with the additional dependency on virtual Machines and MySQL, the set up of ScSF is rather complex. Moreover, ScSF requires a significant amount of resources in the machines where the simulation will be executed.

Batsim [40] is developed on top of the SimGrid simulation framework.[32] Batsim decouples the dispatcher from the simulator and allows it to be implemented in any programming language, yet both the simulator's and the dispatcher's source code and binaries are available only for GNU/Linux. Batsim takes as input a file in a JSON-based format, and provides a script to translate from SWF with which it is possible to read real workload datasets. However, all jobs are loaded in memory at the beginning of simulation which can hinder the performance when experimenting with a large number of jobs. While users can define different resource types as supported by SimGrid, the concept of a single node possessing heterogeneous resources is not natively implemented in the simulator. This calls for significant effort when users wish to model a system using heterogeneous resources. The dispatchers need to be adapted as well in order to take into account the new representation of a system. Similar to AccaSim, additional data regarding the current system status can be used in Batsim for instance, to model the energy consumption of the system. The type of data, however, depends exclusively on the capabilities of SimGrid. And finally, while Batsim includes a workload generator, it is simple, useful for testing purposes only, and is not intended for dispatching research.

Alea [73] is developed on top of the GridSim simulation framework.[33] Job

[31] `https://slurm.schedmd.com/`
[32] `http://simgrid.gforge.inria.fr/`
[33] `http://www.cloudbus.org/gridsim/`

submission, resource management and job dispatching are driven by the predefined workload format, resource types, and dispatchers. The implementation in Java is open-source and cross-platform. However, any customization to the simulator needs to be done at the source code level, which can be complicated and error-prone. PYSS [83, 80, 92] and OCS [52] have similar characteristics to Alea, but provide less advanced WMS features as they are developed primarily for a specific research work in dispatching. In general, simple simulators like PYSS and OCS hinder the design of novel advanced dispatchers and their evaluation which requires a more flexible way to represent a WMS.

In [58], an energy aware WMS simulator, called Performance and Energy Aware Scheduling (PEAS) simulator is described. With the main aim being to minimize the energy consumption and to increase the throughput of the system, PEAS uses predefined dispatchers and workload dataset file format, and the system power calculations are based on fixed data from SPEC benchmark[34] considering the entire processor at its max load. PEAS is available only as GNU/Linux binary, therefore it is not customizable in any of these aspects.

Brennan et al. [24] define a framework for WMS simulation, called Cluster Discrete Event Simulator (CDES), which uses predefined scheduling algorithms and relies on specific resource types. Although CDES allows reading real workload datasets for job submission, it loads all jobs in memory at the beginning of the simulation, like Batsim does. Moreover, the implementation is not available which prevents any form of customization.

In [66], a WMS simulator based on a discrete event library called Omnet++[35] is introduced. Similar to ScSF, only automatically generated synthetic job descriptions are accepted for job submission. Since Omnet++ is primarily used for building network simulators and is not devoted to workload management, there exist issues such as the inability to consider different types of resources as in CDES. Moreover, due to lack of documentation, it is hard to understand to what extent the simulator is customizable.

The main issues presented in the existing WMS simulators w.r.t. to AccaSim can be summarized as complex set up and need of many virtual machines and resources, inflexibility in the workload source and resource types, limited support for additional data, potential performance degrade with large workload datasets, difficulty or the impossibility of the customization of the WMS, platform restriction, and unavailable or undocumented implementation. As AccaSim is developed for facilitating job dispatching research in HPC systems, it is designed to be scalable to large workload datasets and provides maximum flexibility in representing a WMS in terms of workload source, resource types, and dispatchers. It is open-source and cross-platform, simple to install and use, and is easy to customize via abstract class implementations without having to touch the source code.

[34]`https://www.spec.org/power_ssj2008/`
[35]`http://www.omnetpp.org/`

3.4 Comparison of Simulators

In this section, we contrast AccaSim with a critical attention against ScSF, Batsim and Alea which are the most similar simulators to AccaSim.

3.4.1 Comparison to ScSF

ScSF[36] is a complex framework which needs an entire testing environment for running. The environment should have at least two real or virtual machines with dedicated resources, enough hard disk space for the simulator and its components, and external applications such as a database. The network connection is also a key point in the simulation, since it is required to have a low latency in order to maintain a fast link between its components. We do not compare AccaSim to ScSF experimentally for the following reasons. First, the physical resources needed for experimentation with ScSF are much more than those required by AccaSim. Second, the processes involved in a simulation are more complicated, and they are not encapsulated in a single parent process, as in AccaSim, which hinders a fair comparison. For instance, there are processes that are executed in the MySQL database or that depend on ssh connections, which can affect the performance evaluation. Third, job submission in ScSF is performed only by its own workload generator which restricts the experiments to the synthetic jobs generated by ScSF itself.

3.4.2 Comparison to Batsim and Alea

	AccaSim	Alea	Batsim
Workload sources	Customizable	Only SWF	Only JSON (derived from SWF)
Heterogeneous Resources	Customizable	Not possible	Not possible
Dispatchers	Customizable	Customizable at source-code level	Customizable on any programming language
Aditional data	Customizable	Not possible	Customizable (restricted to SimGrid)
Cross-platform	Any supporting Python >= 3.5	Any supporting Java 2	GNU/Linux

Table 3.1: Summary of simulators features.

In Table 3.1, we recall the main differences between AccaSim, Alea and Batsim presented in the related work. Following, we conduct an experimental study to compare the performance of AccaSim to Batsim and Alea using three real

[36]`http://frieda.lbl.gov/download`

workload datasets, which are freely available in SWF. The study is performed on an Ubuntu 16.04 machine with an Intel Core i7-2600 CPU, 16 GB of RAM and a WD10EZEX HDD with 1 TB of capacity. The software used for each simulator experiment are AccaSim 1.0 with Python 3.6.5, Batsim 2.0.0 with Batsched 1.2.0, and finally Alea 4.0 with OpenJDK 1.8.0_171 and 4 GB of max. heap size. All the scripts used to setup and run to experiments, and to evaluate their results are available on the AccaSim GitHub repository.[37]

Workload datasets It is important to compare the simulators' performance on datasets diverse in terms of size and time span, so as to derive robust conclusions on their behavior, especially on how they scale up to large workload datasets. The three datasets on which the experiments are based differ in these aspects. They range from medium-size to very large-size, and they are created in time periods ranging from a decade ago to recent years. We utilized three workload datasets presented in Section 2.1.4. The first workload dataset corresponds to the SETH system, which contains 202,871 jobs. The system was composed of 120 nodes, with 480 cores and 120 GB of RAM in total. The second dataset belongs to the RICC system and contains 447,794 jobs. RICC was composed of 1,024 nodes, 8192 cores and 12 TB of RAM in total. The last workload dataset is based on a workload trace collected from the MetaCentrum system, and it contains 5,731,100 jobs. MetaCentrum was composed by 495 nodes, 8412 cores and 10 TB of RAM in total.

Experimental setup Each experiment corresponds to the simulation of one of the three workload datasets using one of the three simulators. In order to isolate the core actions of a simulator from external factors, such as non-optimal dispatcher implementations, we use a dispatcher which rejects any submitted job. While the rejecting dispatcher is available in AccaSim and Batsim, we implemented it ourselves in Alea. We evaluate the simulators' performance in terms of the total CPU time required to run an experiment and memory footprint. To do so, we use a script which sequentially runs each experiment and repeats it 10 times as a child program in a new process so as to obtain reliable and representative results. The script records each experiment's start and ending time, and gathers the memory consumption every 10ms by using the Python psutil library.[38]

Batsim[39] is conveniently packaged in the Nix package manager for an easy and clean installation on any Linux distribution with superuser privileges. Batsim does not accept SWF in input, and instead provides a script to convert SWF into the required format. This script also works as a workload preprocessor which removes jobs with incomplete or erroneous data. The CPU time and memory consumption of this preprocessing phase is not considered in the Batsim performance result. Instead in AccaSim and Alea, a similar preprocess-

[37] `https://git.io/fhmbM`
[38] `https://pypi.org/project/psutil/`
[39] `https://github.com/oar-team/batsim`

Workload				Simulator		
				AccaSim	Batsim	Alea
Seth	Total time (MM:SS)		μ	**00:15**	00:34	**00:15**
			σ	0.2	0.5	0.5
	Mem. (MB)	Avg.	μ	**18**	596	161
			σ	0.1	2.5	5.4
		Max.	μ	**18**	964	209
			σ	0.1	0.2	23.7
RICC	Total time (MM:SS)		μ	00:27	01:03	**00:24**
			σ	0.5	0.7	0.2
	Mem. (MB)	Avg.	μ	**21**	1,220	162
			σ	0.1	5.4	5.6
		Max.	μ	**26**	2,072	272
			σ	0.1	0.1	52.3
MC	Total time (MM:SS)		μ	**06:23**	29:29	09:08
			σ	4.1	14.2	3.7
	Mem. (MB)	Avg.	μ	**19**	12,647	195
			σ	0.1	137.2	17.4
		Max.	μ	**19**	15,431	1,165
			σ	0.2	6.7	234.4

Table 3.2: Performance comparison of AccaSim, Batsim and Alea.

ing is carried out during job submission, therefore the corresponding CPU time and memory consumption are included in the AccaSim and Alea performance results.

Alea[40] is distributed as a Netbeans Java project in which the entire source code is available. All dependencies and a sample simulation configuration are provided. As opposed to Batsim, Alea accepts SWF in input. However, Alea needs the number of expected jobs in the simulation. Since the number of jobs in the workload may reduce during the preprocessing step, a mismatch with the workload size may crash the job submission process. We indeed faced the problem with the Seth dataset and worked around it by using a number of jobs (200,500), obtained by trial and error, lower than the size of the workload (202,871). Another issue in Alea is that it includes hardcoded instructions for specific datasets or systems which may have to be modified for recent or custom datasets. This kind of implementation makes Alea rather difficult to use.

Experimental results We present the results in Table 3.2, where the MetaCentrum dataset is abbreviated as MC, the total CPU time spent in an experiment is expressed in MM:SS, and the memory usage is expressed with its average and maximum values in MB. The reported values of an experiment are aggregated across all the 10 iterations, and both mean (μ) and standard devi-

[40]`https://github.com/aleasimulator/alea/`

ation (σ) are shown. Across the same dataset and metric, the best results are indicated in bold.

It is clear to see that AccaSim uses up much less memory than the other simulators due to its incremental job loading and job removal capability. This approach is shared by Alea which shows better performance than Batsim. As was discussed in Section 3.3, Batsim loads in memory the preprocessed data from the workload at the beginning of the simulation, which clearly hinders the performance when experimenting with a large workload dataset. As for the total CPU time, AccaSim and Alea show competitive results. Despite AccaSim's more general and costly approach in creating synthetic jobs that can have additional attributes with respect to Alea, the results are close with the medium-size Seth and large-size RICC datasets. AccaSim shows the best results with the very large-size MetaCentrum dataset. Batsim's performance worsens, as the workload size increases. This can be explained by its high memory consumption. In general, when an application requires high amount of memory, the OS has to employ auxiliary data structures at the expense of reduced performance. In addition, Batsim is not optimized for fixed-length job execution models, but rather for models which take into account network and CPU contention.

We can conclude that, Accasim is scalable to large workload datasets, and overall it performs much better than the similar simulators Batsim and Alea.

3.5 Case Study

In this section, we present a case study to illustrate~~'s~~ AccaSim use in job dispatching research. We here focus primarily on dispatcher evaluation and synthetic workload generation. AccaSim can as well be used to develop advanced dispatchers, see [51] for an example. We leave further examples of dispatcher development in AccaSim to future work.

The experimental study conducted in this section is performed on a CentOS 7.3 machine with two Intel Xeon E5-2630 v3 CPUs, 128GB of RAM, using Python 3.6.5 and Accasim 1.0. All the scripts used to setup and run the experiments, and to evaluate their results are available on the AccaSim GitHub repository. [41]

3.5.1 Experimental setup for dispatcher evaluation

To conduct the experimental study regarding dispatcher evaluation, we use the Seth dataset introduced in Section 3.4, given its reasonable size for proof of concept. The corresponding synthetic system configuration is shown in Figure 3.6. Since multiple jobs can co-exist on the same node, we consider a better representation of the system, made of cores instead of processors. We note that AccaSim can as well be used to simulate an HPC system possessing heterogeneous resources, such as the Eurora system, as was shown in [93].

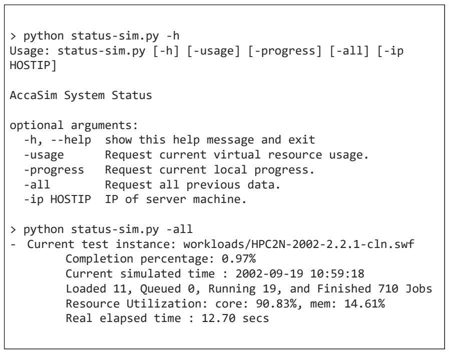
[41] `https://git.io/fhmba`

```
{
        "system_name": "Seth − HPC2N",
        "start_time": 1027839845,
        "equivalence": {
                "processor": {
                        "core": 2
                }
        },
        "groups": {
                "g0": {
                        "core": 4,
                        "mem": 1000000
                }
        },
        "resources": {
                "g0": 120
        }
}
```

Figure 3.6: System configuration of Seth.

As for dispatchers, we employ all the implemented and available dispatchers of AccaSim which are composed of all combinations between the schedulers: First In First Out (FIFO), Shortest Job First (SJF), Longest Job First (LJF) and Easy Backfilling with FIFO priority (EBF); and the allocators: First Fit (FF) and Best Fit (BF). To run the experiments, we conveniently use the *experimentation* tool of AccaSim, as was shown in Figure 3.4. Each experiment corresponds to the simulation of the Seth workload using a specific dispatcher, and is repeated 10 times so as to obtain reliable and representative results.

3.5.2 Dispatcher evaluation



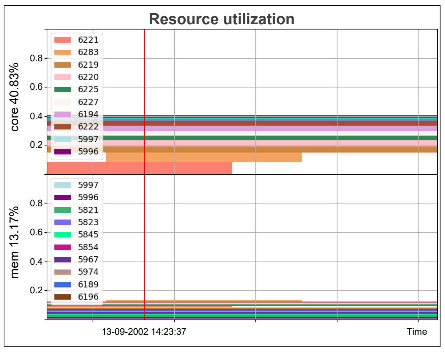
```
> python status-sim.py -h
Usage: status-sim.py [-h] [-usage] [-progress] [-all] [-ip
HOSTIP]

AccaSim System Status

optional arguments:
  -h, --help  show this help message and exit
  -usage      Request current virtual resource usage.
  -progress   Request current local progress.
  -all        Request all previous data.
  -ip HOSTIP  IP of server machine.

> python status-sim.py -all
-  Current test instance: workloads/HPC2N-2002-2.2.1-cln.swf
        Completion percentage: 0.97%
        Current simulated time : 2002-09-19 10:59:18
        Loaded 11, Queued 0, Running 19, and Finished 710 Jobs
        Resource Utilization: core: 90.83%, mem: 14.61%
        Real elapsed time : 12.70 secs
```

(a) System status.

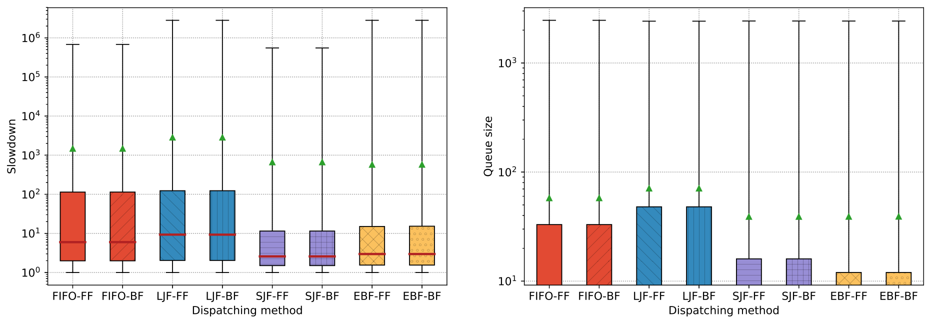


(b) System visualization.

Figure 3.7: Monitoring tools.

Dispatchers can be evaluated and compared from different perspectives thanks to AccaSim's tools and output data. In Figures 3.7a and 3.7b, sample snapshots taken by the two components of the *monitoring* tool at certain time points during the FIFO-FF experiment are shown. The *system status* tool receives command line queries to show a variety of information regarding the current synthetic system status, such as the queued jobs, the running jobs, the completed jobs, resource utilization, the current simulation time point, as well as the total CPU time elapsed by the simulator. The *system visualization* tool summarizes the allocation of resources by the running jobs each indicated with a different color, using an estimation (such as wall-time) for job duration. The display is divided by the types of synthetic resources. In our case study, the core and memory usage are shown separately.

The *experimentation* tool automatically generates plots to compare the dispatchers according to their effect on system utilization, job response times, system throughput, and their performance in terms of the time they incur for generating a decision. For job response times and system throughput, two metrics are used. The first is the job slowdown, a common indicator for evaluating job scheduling algorithms [43], which quantifies the effect of a dispatching method on the jobs themselves and is directly perceived also by the HPC users. The slowdown of a job j is a normalized response time and is defined as $slowdown_j = (T_{w,j} + T_{r,j})/T_{r,j}$ where $T_{w,j}$ is the waiting time and $T_{r,j}$ is the duration of job j. A job waiting more than its duration has a higher slowdown than a job waiting less than its duration. The second metric is the queue size, which counts the number of queued jobs at a certain dispatching time. This metric is a measure of the effects of dispatching on the computing system itself. The lower these two metrics are, the better job response times and system throughput are.



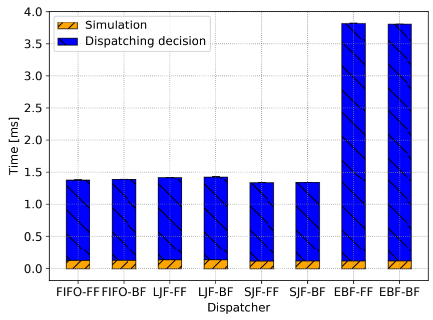
(a) Distributions for job slowdown.

(b) Distributions of queue size.

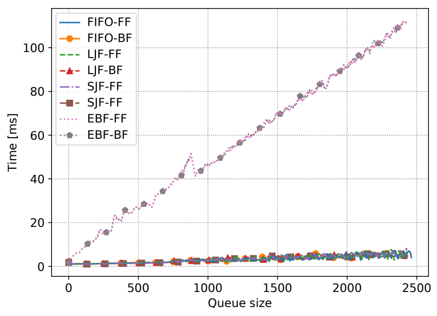
Figure 3.8: QoS evaluation.

In Figures 3.8a and 3.8b, we present the automatically-generated box-and-whisker plots showing the distributions of the slowdown and the queue size for each experiment. We can see that SJF and EBF-based dispatchers achieve the best results, independently of their allocators probably due to the homogeneous nature of the synthetic system. Their slowdown values are mainly lower than the median of the FIFO and LJF-based dispatchers. SJF maintains overall lower slowdown values than the others, but a higher mean than the EBF. SJF maintains also slightly higher mean in the queue size than the EBF. The scheduling algorithm of EBF does not sort the jobs, like SJF, instead it tries to fit as many jobs as possible into the system, which can explain the best average results achieved in terms of slowdown and queue size.

In Figure 3.9a, we present the automatically-generated plot which shows the average CPU time required at a simulation time point for each dispatcher. The average CPU time of an experiment is obtained by aggregating the data from all its 10 iterations. The time spent in simulation, other than generating the dispatching decision, is constant (around 0.2 ms) across all the experiments, and the EBF-based dispatchers spend much more time in generating a decision than the others. In Figure 3.9b, we instead present the automatically-generated plot that analyzes the scalability. Specifically, it reports for each queue size the average CPU time spent at a simulation time point in generating a dispatching decision. Also in this case, we considered the data related to all 10 iterations of the experiments. While all the dispatchers scale well, the EBF-based dispatchers require more CPU time for processing bigger queue sizes, due to their scheduling algorithm which tries to fit as many jobs as possible into the system.



(a) Average CPU time at a simulation time point.



(b) Average CPU time at a simulation time point to generate a dispatching decision w.r.t. queue size.

Figure 3.9: Timing of the dispatchers.

AccaSim users are free to analyze the output data as they wish to evaluate the dispatchers further. For instance, to compare in more detail the dispatchers' performance, they can extract the total usage of CPU time and memory of each experiment, as reported in Table 3.3. In the table, the time columns correspond to the total CPU time spent by the simulator and the time spent in generating the dispatching decision; whereas the memory columns give the average and the maximum amount of memory utilized over the entire simulation time points. The reported values of an experiment are aggregated across all the 10 iterations, and both mean (μ) and standard deviation (σ) are shown.

Most of the experiments took around 8 minutes. The exceptions are the EBF-based experiments which require around 22 minutes because the underly-

Dispatcher	Time (MM:SS)				Memory (MB)			
	Total		Disp.		Avg.		Max.	
	μ	σ	μ	σ	μ	σ	μ	σ
FIFO-FF	08:01	2.6	07:15	2.3	76	0.2	82	0.3
FIFO-BF	08:05	1.8	07:18	1.6	79	0.1	85	1.1
LJF-FF	08:13	2.4	07:24	2.1	80	0.7	86	0.9
LJF-BF	08:17	2.3	07:27	2.1	81	0.8	86	0.9
SJF-FF	07:46	2.2	07:04	2.0	82	0.8	86	0.5
SJF-BF	07:49	1.7	07:06	1.5	82	0.4	86	0.6
EBF-FF	22:24	2.9	21:41	2.7	82	0.6	85	0.7
EBF-BF	22:19	4.6	21:36	4.2	82	0.6	84	0.8

Table 3.3: Total CPU time and memory usage during the simulation.

ing dispatching algorithms are computationally more intensive. In accordance with Figure 3.9a, the time spent by the simulator, other than generating the dispatching decision, is constant (around 40 seconds) across all the experiments. The total CPU usage is thus highly dependent on the complexity of the dispatcher. The average memory usage is around 80MB with a peak at 86MB across all the experiments.

Our analysis restricted to the considered dataset reveals that, while the EBF-based dispatchers give the best results in terms of response times and throughput, they are much more costly in generating a dispatching decision. Simple dispatchers based on SJF are valid alternatives with their excellent scalability and with their comparable results in response times and throughput.

3.5.3 Synthetic workload datasets

In order to generate synthetic workload datasets, and later for comparison purposes, we utilize the Seth and RICC datasets introduced in Section 3.4. With each, we generate four datasets using different configurations in terms of resource type, processing unit performance, and the number of jobs. The first dataset includes 50,000 jobs and a 1.5x improvement in core performance. The second includes 100,000 jobs with double number of nodes. The third includes 200,000 jobs, two GPU accelerator cards for a quarter of the nodes with a performance of 933 GFLOPS per second. Finally, the last includes 500,000 jobs, two GPU accelerator cards for a half of the nodes with a performance of 933 GFLOPS per second and a 1.5x improvement in the core performance. The improved performance and the change in the number of nodes are relative to the system that the workload dataset in consideration belongs to. In the following, we first briefly describe the generation process, and then show the similarity between the real and the generated datasets.

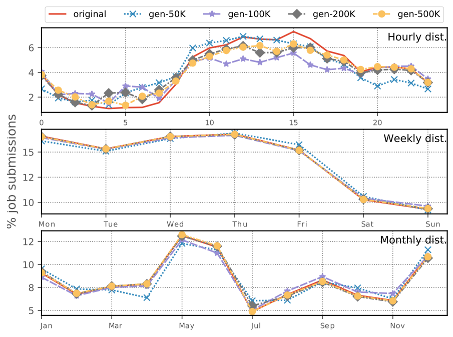
Synthetic workload dataset generation. The first aspect to compare between a real and a synthetic workload dataset is the job submission cycle which refers to the job submission times and reflects the usage of the system by its users. The cycles could be represented by certain periods of working time to reflect better the real usage of the system. The *WorkloadGenerator* calculates the submission time of a job j based on a daily cycle model proposed in [85]. In the original algorithm, named Slot Weight Method, a day is represented by 48 slots of 30 minutes each (s). Thus, the first slot starts at midnight, the next one at 00:30, and so on. Each slot has a specific weight which is the ratio between the number of jobs belonging to the time slot and the total number of jobs in the real workload dataset, which represents a measure for selecting a slot for j. The algorithm generates a random value v between 0 and 5 to represent the maximum number of days that can elapse between j and its predecessor, based on the statistical distribution of the interarrival times of the real workload dataset. For selecting a slot, the algorithm starts from the slot of the predecessor of j. The slots are considered as a circular list. For each considered slot, if v is greater or equal to the slot weight, v is updated by subtracting the slot weight. Update continues with the next slot, otherwise, the algorithm stops and selects the current slot. Then, the job submission time of j is calculated by summing the half hours of all the surpassed slots plus the remaining amount of v.

We modify this algorithm in two aspects so as to assimilate a real job submission cycle. First, we modify the fixed upper-bound v_{max} of v to the maximum value of the interarrival times of the dataset. Second, we add a dynamic process that modifies v_{max} during a job submission time generation. For this purpose, we calculate the ratio between the number of the currently generated jobs and the required jobs in three different ways in relation to the last submitted hour, the last submitted day, and the last submitted month. This allows to keep the generation of values as similar to the real data as possible. Then, we calculate the progress ratio of each ratio by dividing it by the respective ratios in the real data. The overall progress ratio is the multiplication between all progress ratios (pr). Finally, v_{max} is dynamically adapted at each job submission time generation as follows:

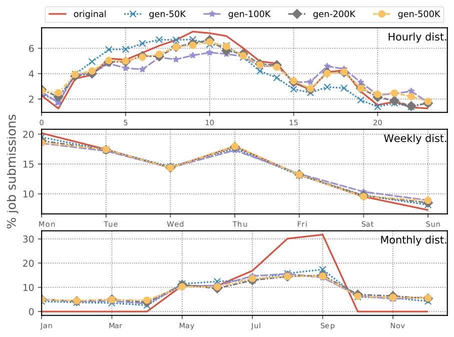
$$v_{max} \leftarrow v_{max} - (v_{max} - s) * (1 - pr)$$

If $pr = 1$, the job submission time generation of the predecessor reached the real ratios, thus for j, we use v_{max}. In addition, when the real data does not include specific months, pr has only hourly and daily ratios.

The second aspect to compare is the theoretical computed FLOPs for each job during its execution in the system, which depends on, among others, its duration and resource requests in terms of resource type restricted to the processing units (e.g., cores, GPU, MIC, etc.) and quantity. These features of a job are generated in three phases. The first phase is based on an algorithm from [85] to select the job type, serial or parallel, and the number of requested nodes. Since this algorithm considers a job parallel if it runs on multiple nodes, we modify it to create parallel jobs on a single node, i.e. when the number of required cores is greater than one. In the second phase, the resource request is defined by randomly choosing among the available resource types and assigning them a quantity, using a uniform distribution and considering the request limits passed as an argument during the *WorkloadGenerator* instantiation, as shown in Figure 3.5. Finally, in the third phase, the job duration is calculated as the division between (i) a random FLOP value and (ii) the dot product of the resource requests and their corresponding theoretical performance, multiplied by the number of required nodes.



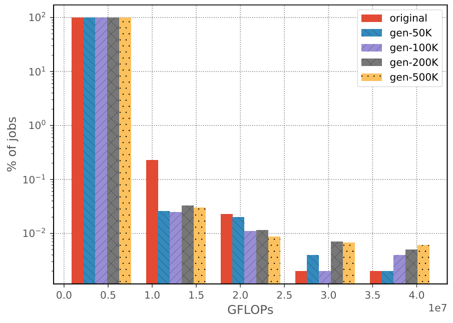
(a) Seth.



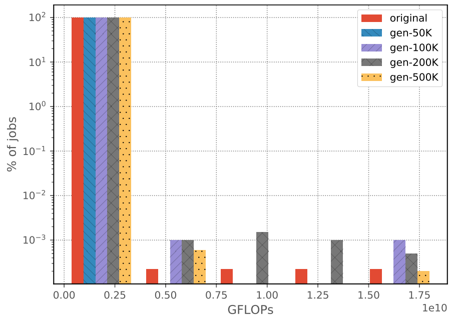
(b) RICC.

Figure 3.10: Workload datasets

Comparison to the real workload datasets. Figures 3.10a and 3.10b show the the hourly, daily, monthly job submission distributions of the real and the generated workload datasets. The introduced modifications generate submissions that took place mainly during the working hours, weekdays, and working months, resulting in a more realistic scenario. The generated datasets look very similar to the real datasets, except in the case of the monthly distribution of the RICC dataset. The reason is that the RICC job submissions span to five months, not to an entire year.



(a) Seth.



(b) RICC.

Figure 3.11: Workload datasets

Figures 3.11a and 3.11b show the distributions of the computed theoretical FLOPS, here represented in GFLOPS, between the real and the generated workload datasets. We observe a similar pattern also here. The usage of the FLOPS calculation for the generation of the jobs' features allows maintaining a distribution similar to the real workload dataset, independent of the configuration of the real system. In this way, the real dataset can be tested with other system configurations using the generated dataset.

3.6 Summary

We introduced AccaSim, a library for simulating WMS in an HPC system, which offers to the researchers an accessible tool to facilitate their job dispatching research. The library is open-source, implemented in Python, which is freely available for any major operating system, and works with dependencies reachable in any distribution. It is executable on a wide range of computers thanks to its lightweight installation and light memory footprint. AccaSim is scalable to large workload datasets and provides support for easy customization, allowing to carry out experiments across different workload sources, resource types, and dispatching algorithms. Moreover, AccaSim enables users to develop novel advanced dispatchers by exploiting information regarding the current system status, which can be extended for including custom behaviors such as energy and power consumption and failures of the resources. Last but not least, AccaSim aids users in their experiments via automated tools to generate synthetic workload datasets, to run the simulation experiments and to produce plots to evaluate dispatchers. The researchers can thus use AccaSim to mimic any real system, including those possessing heterogeneous resources, develop advanced dispatchers using for instance power and energy-aware, fault-resilient algorithms, and test and evaluate them in a convenient way over a wide range of workload sources by using real workload traces or by generating them.

In order to highlight the main contributions of AccaSim, we discussed the existing related simulators, presented a critical comparison to the most similar simulators, and showcased AccaSim's use in job dispatching research, specifically in dispatcher evaluation and synthetic workload generation. In future work, we plan to use AccaSim to develop advanced dispatchers using power and energy-aware, fault-resilient algorithms.

Chapter 4

Job duration prediction in HPC systems

The duration of jobs was introduced as an essential asset to improve the quality of dispatching decisions. However, this value is unknown at dispatching time. Instead, usually a highly overestimated prediction in place of the actual duration is known. Dispatchers often use this value as the expected job duration, which can be either the user expected duration or the default wall-time of the queue. From a system management point of view, this value is the maximum time of a job is allowed to be executed on the system.

Therefore, in order to improve the quality of decisions, we try to improve the expected job duration using a better prediction method. Our motivation is to study whether transforming the log data produced by an HPC system into useful knowledge about its workload may produce better dispatching decisions. Thus, in Section 4.1, we propose a data-driven job duration prediction that uses historical data to construct user-job profiles. In addition, we analyze the effect of different prediction approaches in making dispatching decisions. Next, in Section 4.2, we propose a possible improvement of the previous prediction method, maintaining its simplicity but based on other observations. Both prediction methods aim to improve dispatchers who base their algorithms on job durations.

Section 4.1 corresponds to the publication [51].

4.1 Data-driven job dispatching in HPC systems

Duration of jobs is an important consideration in dispatching decisions and knowing them at job submission time clearly facilitates better algorithms. Dispatching algorithms are often developed with the assumption that job durations are known [47, 29]. Even if this is not practical, in some cases it may be possible to rely on user-provided estimates of job duration [47, 18]. Many HPC systems allow users to define a wall-time value, and use a default value when users fail to provide one. This wall-time can be considered a crude prediction of job duration, which in the case of Eurora is set on a per-queue basis.

It has been shown that in general user estimations are not reliable [47], while predefined wall-times are inflexible to account for all user needs. In these conditions, prediction of job duration through other means may prove to be an important resource. Here, we describe a simple data-driven heuristic algorithm that relies on user histories to predict job duration. The data-driven approach is particularly useful when user data can be stored for longer periods of time, which is increasingly feasible through modern Big Data tools and techniques.

Algorithm 2: Data-driven prediction method.

```
1 $d_i^{'} \leftarrow$ DD-Prediction_1(user, job$_i$, database)
2 wtime $\leftarrow$ job$_{wtime}$
3 if $user \notin database$ then return wtime
4 matched_job_$j$ $\leftarrow$ Rules(user, job$_i$, database)
5 if $matched\_job\_j <> NULL$ then
6     $d_j^r \leftarrow$ d(matched_job_$j$)
7     return min$(wtime, d_j^r)$
8 end
```

Our heuristic constructs job profiles from the available workload data. The profile includes job name, queue name, user-declared wall-time, and the number of resources of each type (CPU, GPU, MIC, nodes) requested. Each user is analyzed separately.

Prediction is based on the observation that jobs with the same or similar profiles have the same duration for long periods of time — there is a temporal locality of job durations. Then, at some point, the duration changes to a new set of values, which are again stable in time. This could be due for instance to changes in user behavior: a user first tests the code with short runs, then decides to run the real simulation which may last longer, then may decide to test again after having made changes, and so on. Another explanation could be switching between input datasets: the user performs repeated runs on one set of data, then moves to another.

Algorithm 2 shows the pseudo-code of our proposal for the data-drive prediction method. For each job i, our heuristic searches for the last job j with a similar profile, and uses the duration of that job $d_j r$ to predict the duration $d_i^{'}$ of the new one. In line 3, the algorithm verifies if the user already exists in the *database*, if it does not exists it return the requested wall-time. The database is updated after each job completion, inserting the job data required by the profiles, identical profiles are updated.

We analyze users separately in line 4. The similar profile is identified using a set of consecutive rules. First, a full profile match is searched for, then if this does not exist in the user history, a profile where the job name has the same prefix is looked up. This follows from the observation that users often name jobs with similar durations with the same job name followed by a number (e.g. “job1”,“job2”). If this is unsuccessful, we allow for resources used to differ, as long as the full job name, queue and wall-time are the same. If also this search fails, we look for the same match but with the name prefix rather than the exact name. If none of these rules give a match, we look for the last job with the same name, or, as a last resort, the same name prefix. If all rules fail, then we take the wall-time as the predicted duration. In all cases, the prediction is capped by the wall-time as defined in line 7.

Machine learning techniques may not be satisfactory in comparison with our simple heuristic. We believe this is due to the temporal locality observed in the data, and also due to the fact that jobs with the same profile may have several different durations depending on when they were submitted. This means that a regular regression model would try to fit a wide range of values with the same features, resulting in an averaging of the observed durations.

4.1.1 Experimental study

We perform two experiments regarding the use of the job duration prediction. First, we evaluate the accuracy of the prediction based on the Eurora workload dataset (Section 2.1.4), i.e. performed off-line using the dispatching decision of its original dispatcher. Second, we perform an extensive experimental study to evaluate the usage of the previous prediction in a simulated environment. The simulated environment is provided by AccaSim, Chapter 3, to simulate the Eurora system with the workload trace described in Section 2.1.4. This study considered five dispatching methods described in Section 2.1.3. Since in an HPC system, scheduling goes hand in hand with allocation in order to perform job dispatching, we combined *Shortest Job First* (SJF), *Longest Job First* (LJF), and *Easy Backfilling* (EBF) with the First-Fit allocation policy. Instead for *Priority Rule-Based* (PRB) and *Hybrid Constraint Programming dispatcher* (`HCP`), we combined with the Best-Fit allocation policy. In all methods, the objective of the dispatcher is to minimize the total waiting time of the submitted jobs. The waiting time of a job is the time passed between its submission and its starting time. Therefore, for each of the five dispatching methods, there are three estimations of job duration, resulting in 15 combinations (e.g., for the SJF method we have SJF-W, SJF-D and SJF-R corresponding to wall-time prediction, data-driven prediction and real duration, respectively).

The dispatchers are used together with three estimations of job duration: prediction based on wall-time (W), data-driven prediction presented in Section 4.1 (D) and real duration (R). The real duration was included to provide a baseline to which the other two predictions are compared.

AccaSim library already includes the implementations of the SJF, LJF and EBF dispatching methods. The PRB and `HCP` implementations are available for download in the AccaSim website [42]. All experiments were ran on a CentOS machine equipped with Intel Xeon CPU E5-2640 Processor and 15GB of RAM.

[42]`http://accasim.readthedocs.io/en/latest/`

4.1.2 Prediction performance results

We evaluate the mean absolute error (MAE), which is a common measure of forecast error in time series analysis. The MAE is an average of the absolute errors $\sum_{i=1}^{n} \frac{|e_i|}{n} = \frac{|y_i - x_i|}{n}$, where y_i is the prediction and x_i is the true value. The MAE of the heuristic and the wall-time approach with respect to the real duration were shown to be 40 mins and 225 mins, respectively. The heuristic prediction shows thus an improvement of 82% over the wall-time approach. In Figure 4.1, we show the empirical cumulative distribution function (ECDF) of the prediction accuracy $A = d_i^r / d(\tau_i)$, the ratio between the real and the predicted duration of a job, of all the three methods. The empirical ECDF shows the proportion of scores that are less than or equal to each score of A on Eurora. When $A = 1$, the duration $d(\tau_i)$ matches the real duration d_i^r. We have underestimation when $A > 1$, overestimation when $A < 1$. In theory, we should not have underestimation with the wall-time approach because in a real system a job is killed if it takes longer than its d_i. However, a system requires extra time after a job is killed or completed to bring the resources on-line again and this extra time is reflected to the dataset. Therefore, in some cases we have $A > 1$ in Figure 4.1. We have $0.75 \leq A \leq 1.25$ for about 50% of the workload with the heuristic, and for less than 10% with the wall-time approach. On the other hand, the heuristic introduces considerable underestimation. The exact under and overestimation rates are 3.6% and 96.3% for the wall-time and 25.8% and 53.7% for the heuristic, respectively.

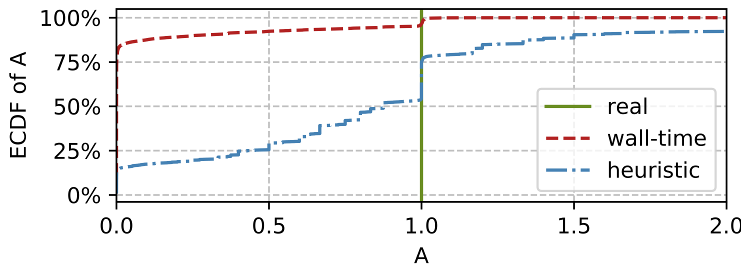


Figure 4.1: The distribution of the accuracy of the three prediction methods.

Figure 4.2 shows the distribution of the absolute errors for wall-time and the data-driven prediction, showing that in the data-driven case, these are concentrated towards small values, while in the case of the wall-time the distribution peaks at errors over 1 hour. The plot shows clearly that our data-driven prediction produces much better results compared to wall-time prediction.

4.1.3 Dispatching performance results

To compare the quality of the dispatching decisions of the 15 combinations, we have selected two criteria. The first is *job slowdown*, a common metric for evaluating job scheduling algorithms[43], which quantifies the effect of each

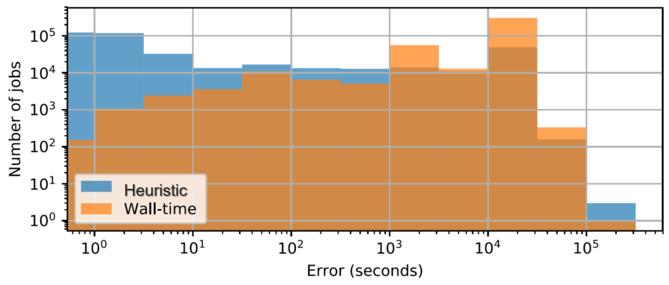


Figure 4.2: Absolute data-driven prediction error, compared to wall-time prediction.

method on the jobs themselves and is directly perceived also by the HPC users. Slowdown of a job j is a normalized waiting time and is defined as $slowdown_j = (T_{w,j} + T_{r,j})/T_{r,j}$ where $T_{w,j}$ is the waiting time and $T_{r,j}$ is the duration of job j. A job waiting more than its duration has a higher slowdown than a job waiting less than its duration. The second criterion is the *number of queued jobs* at a given time. This metric is a measure of the effects of dispatching on the computing system itself, being directly related to system throughput: the lower the number of waiting jobs, the higher the throughput.

To analyze the effects of prediction on job dispatching, we plot the distribution of our evaluation criteria for all 15 combinations of the dispatching methods and duration predictions. For easy visualization of distributions, we use boxplots that show the minimum and maximum values (top and bottom horizontal lines), the range between the 1st and 3rd quartiles (the colored box), the median (horizontal line within the box) and the mean (the triangles). Note that with the logarithmic scale on the vertical axis, some of these elements may be missing from the plots, meaning their value is zero.

Effects of prediction on jobs. The first analysis looks at job slowdown for all 372,321 jobs dispatched. Figure 4.3 shows the distribution of slowdown achieved by each dispatching method with each prediction type. For better visualization, we plot only the jobs where slowdown is different from 1 in at least one method-prediction combination. The removed jobs are those that are dispatched immediately as they arrive in the system, so are not relevant for our comparison. As the figure shows, the dispatching methods displaying best performance when the most basic and least effective prediction is used (wall-time) are PRB and CPH, while the methods performing worst are LJF and SJF. This is understandable since the latter methods are quite simple while the former employ more sophisticated reasoning.

An interesting effect when using real duration is that not all dispatching methods show a clear benefit. While we observe a clear decrease in slowdown in SJF, EBF and `HCP`, for LJF a significant increase in slowdown is present, while for PRB no change is observed. We understand that prediction does not always help the dispatching methods. One possible explanation is that the incomplete

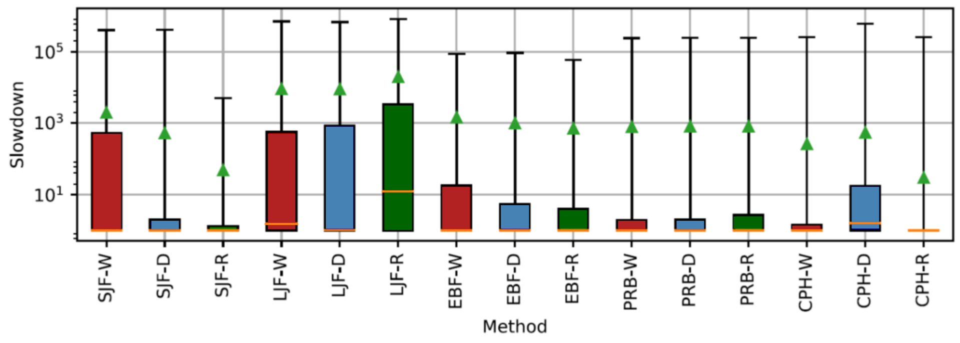


Figure 4.3: Distribution of job slowdown for each method.

nature of the dispatching methods tends to lead to suboptimal decisions which can sometimes be compensated by underestimation of job durations, which will not be possible anymore with a (perfect) prediction.

When using our data-driven prediction in the dispatching methods, we expect the performance to stay between the wall-time prediction and the real job duration. Figure 4.3 shows that this is true for most methods. In the cases of SJF and EBF, the real job duration improves the results, so does our prediction, albeit less effectively. In the case of LJF, real job duration worsens the results, so does our prediction, but less severely. PRB, which already does not benefit from real job duration, does not benefit from our prediction either. The only dispatching method where the performance improves with perfect prediction but decreases with our prediction is `HCP`. We believe this is because our data-driven prediction may sometimes underestimate job duration, which is never the case for wall-time and the real duration. `HCP` is not resilient to job duration underestimation, hence an imperfect prediction can actually be detrimental.

Even if PRB and `HCP` provide the best overall results, we observe that SJF comes in very close, with comparable slowdown, when adding prediction. However, the first two methods are more sophisticated and incur an overhead when building the dispatching decisions, while SJF is a very simple strategy. Hence, in the presence of predictions, one may prefer to use a simple method such as SJF over the heavier methods such as PRB and `HCP`.

To better understand the effects of prediction, we also look at the different job classes. Figure 4.4 shows box-plots of slowdown distributions for short, medium and long jobs. When prediction is beneficial, we see that the jobs that benefit most are the short ones. This is good news, given that a large number of our jobs are short, as we saw in Section 2.1.4. Some smaller differences are also visible on medium jobs, while on long jobs the methods seem to be quite comparable, with slightly larger slowdown values in `HCP` and SJF compared to the rest.

Effects of prediction on the system. Besides effects on individual jobs, it is important to understand prediction's role in improving system-level behav-

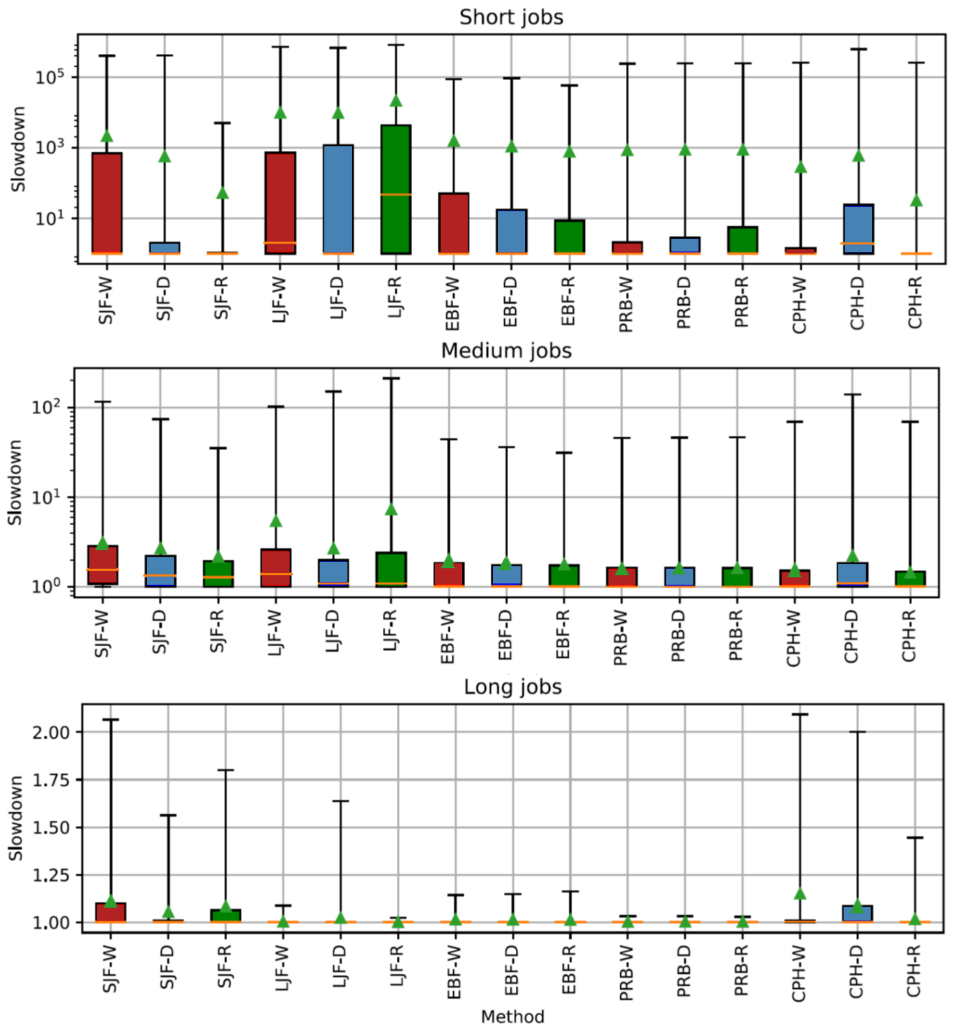


Figure 4.4: Distribution of job slowdown for short, medium and long jobs for each method.

ior. For this, we look at the size of the waiting queue. Figure 4.5 shows the distribution of the number of jobs in the queue at every second. We removed from the plot those time points where there were no jobs in the queue for any of the 15 combinations, because these corresponded to low system utilizations and have no value for our comparison. The figure shows that the effect on the system is similar to the performance measured by the slowdown. In particular, SJF and EBF are improved by prediction (both data-driven and real durations). PRB shows no difference, however queue size is already the shortest among all dispatching methods. LJF does not benefit from prediction, while `HCP` seems to be improved only by perfect and not by our data-driven prediction.

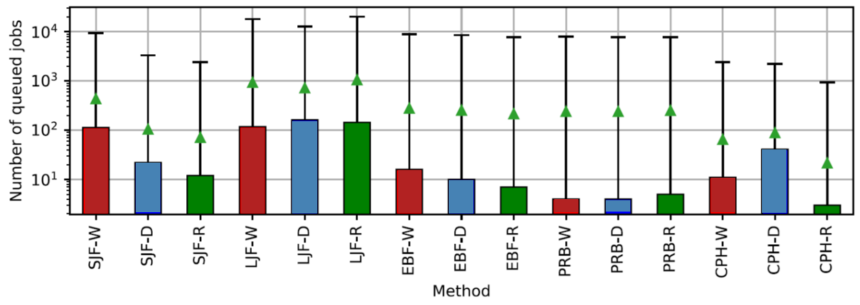


Figure 4.5: Distribution of number of jobs waiting at every second, for each method.

4.2 Improving the data-driven job duration using the user confidence

In the previous Section 4.1, we showed that the throughput of a system, specifically on Eurora, but theoretically applicable for all systems, can be improved by incorporating a job duration prediction method to be used in place of the wall-time as the expected duration of jobs. Previously, we mentioned that, in general, users tend to overestimate the duration of jobs and request more time (wall-time) than the required. Although it is a general observation, almost all users do that. However, some users request just the necessary wall-time, probably because they have the expertise to predict better the duration of jobs, and the data-driven heuristic, from Section 4.1, punishes their knowledge. In this section, we propose an enhancement for the data-driven heuristic, on which we consider the user confidence as a valuable asset during prediction.

It is worth mentioning, although the results are promising, in the later chapters, we do not use this new version because this enhancement was proposed after concluding the experiments of the last chapter. Besides, we do not compare against any dispatcher because we already know that a dispatcher that can be benefited by accurate prediction present better results.

The workload dataset from the Eurora system has 360 users with at least one submission to the system, of which 92% made multiple submissions. For those users, we evaluated their accuracy in terms of requested wall-time (d_i) with respect of the real duration d_i^r (d_i^r/d_i). In Figure 4.6, we show the first 20 users sorted incrementally by the median of their accuracy in percentages. Red lines represents an interval between the perfect prediction $d_i^r \pm 25\%$, i.e. when $d_i = d_i^r$.

In spite of medians neither averages are in the $\pm 25\%$ interval, some wall-time requests are in that interval, thus presenting a good accuracy. It is worth to mention that medians and averages are around 50%, meaning $d_i \pm 50\% \approx d_i^r$, and we can assume at some point users improve their accuracy on the wall-time requests. With this assumption, we propose to benefit users with accurate wall-time requests by considering the record of accuracy in our prediction method.

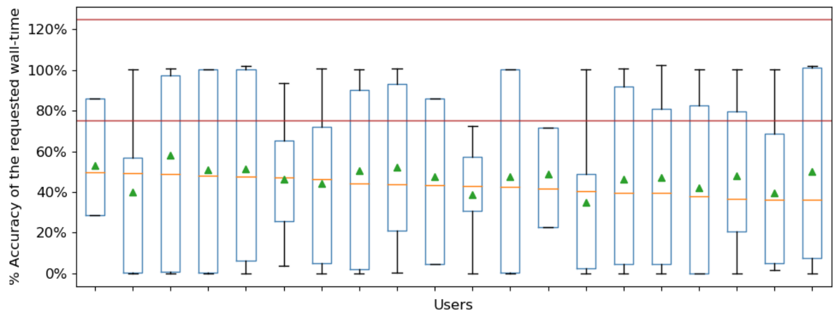


Figure 4.6: Distribution of the accuracy regarding wall-time requests of 20 users of the Eurora system.

Algorithm 4.2 presents the new modifications to the original one. Line 6, returns immediately the requested wall-time if the user with a mean accuracy_record greater or equal than the *threshold*. We notice that in addition to keep the data regarding of completed jobs, also will be necessary to maintain the accuracy_record for each user.

So after a job is completed, the data required by the rules is stores in the database, and the stack of the user accuracy is updated:

$$\text{accuracy\_record}_u = \text{accuracy\_record}_u \cup d_i^r / d_i$$

Algorithm 3: Data-driven prediction with the user confidence method.

```
1  $d_i^{'}$ ← DD-Prediction_2(user, job$_i$, database, accuracy_record)
2  wtime ← job$_{wtime}$
3  **if** *user* $\notin$ *database* **then**
4      **return** wtime
5  **else**
6      **if** MEAN(*accuracy_record*) $\geq$ *threshold* **then return** wtime
7  **end**
8  matched_job_$j$ ← Rules(user, job$_i$, database)
9  **if** *matched_job_j* <> *NULL* **then**
10     $d_j^r$ ← d(matched_job_$j$)
11     **return** min($wtime, d_j^r$)
12 **end**
```

4.2.1 Experimental study

To evaluate the significance of the proposed modifications to our data-driven prediction method, we conducted an experimental study. We evaluate our data-driven prediction with the user confidence method using the entire Eurora workload, considering 404,881 jobs. No dispatcher will be used in this experiment, so we will recreate the PBS decisions. We will consider the last three accuracy records to calculate the mean, and the threshold value is 80%.

We evaluate the MAE and the prediction accuracy A, previously defined in Section 4.1.2, to compare the accuracy of the new prediction method (D2) with respect two others: the wall-time prediction (W), our original version of the prediction method (D1) and the last two prediction method [129] (L2). L2 proposes a model that uses the run times of the last two jobs to predict the duration of the next job.

4.2.2 Accuracy of predictions

Next, we evaluate the accuracy of predictions generated by the user with her requested wall-time (W), our first version of the data-driven prediction (D1), the last two prediction method (L2) and our new proposal (D2). We measure of difference between two paired variables using the MAE metric, the actual duration of a job d_i^r and the predicted one d_i, detailed in Table 4.1.

Predictor	MAE
W	225 min.
D1	39 min.
D2	10 min.
L2	227 min.

Table 4.1: MAE results of W, D1, D2 and L2 prediction methods.

The MAE of the D1 and the W approach with respect to the real duration were shown to be 39 mins and 225 mins, respectively. The D1 shows thus an improvement of 82% over the W, whereas D2 presents an increment of its previous version of 96% with a MAE of 10 min. Instead, L2 increments the MAE with respect to W. Since MAE is an average of the absolute errors, o differentiate better the gains, we show the empirical cumulative distribution function in Figure 4.7.

L2 slightly improves the W predictions, as opposed to the MAE results, by reducing the underestimation ($A > 1$) and overestimation ($A < 1$). D2 shows an improvement of its prior version, with a higher overestimation reduction, only 10% of the predictions are below of $d_i \approx d_i^r + \%50$, and a small reduction in the underestimation. Meaning that more predictions are closer to the perfect prediction ($A = 1$); thus, in general, the results are highly improved.

The user confidence at job submissions seems to be a valid option, at least for the Eurora system, to consider when predicting job durations especially for those that request only the necessary time for running their jobs.

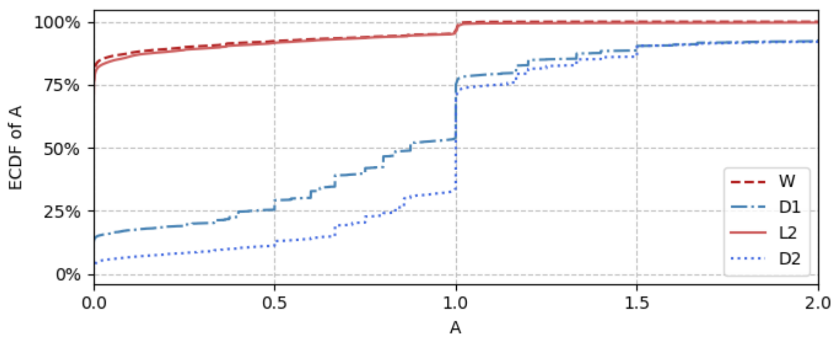


Figure 4.7: The ECDF of the accuracy of W, L2, D1 and D2.

4.3 Summary

A job duration predictor seems to be a necessary resource to develop intelligent dispatchers, given that, in general, dispatchers assume job durations are known in advance and their decisions rely on the expected duration of jobs. Our data-driven job duration predictors are simple and do not produce overhead neither needs big datasets to operate and principally improve user estimations. The first version of the job duration prediction was studied in conjunction with five different dispatching methods, showing and improvement specially for short jobs. Given the prominent presence of short jobs in typical HPC [112] and cloud system [101] workloads, our conclusions should apply to large-scale computational infrastructures in general.

Nevertheless, we improved the results obtained by applying our first version of the data-driven job prediction method to the Eurora workload, we realize that we cannot generalize the user behavior, therefore we incorporate the confidence of users. Thus, users that show a high accuracy in their wall-time request should be rewarded. This new version of our data-driven job prediction method maintains its simplicity and low overhead by including only the data regarding users' accuracy. Given that we improved the accuracy of our predictions, the performance of dispatchers that use our new version of the predictor may also improve their results.

Chapter 5

CP-based dispatchers for heavy and short job-dominated workloads in HPC systems

In this chapter, we propose new dispatchers, which are able to reduce the time required for generating on-line dispatching decisions significantly, and can make effective use of job duration predictions to decrease waiting times and job slowdowns, especially for workloads dominated by short jobs. We build on `PCP` and `HCP`, introduced in Chapter 2.3, and redesign their main components as to tackle their issues introduced in this dissertation. First, we revisit their model and search control mechanism so as to make them resilient to heavy workloads and applicable to on-line dispatching. Second, we study the use of job duration prediction, instead of the expected duration, when generating dispatching decisions. We discuss why naively replacing the expected duration with a predicted duration may be ineffective, if not detrimental for QoS. Consequently, we adapt the model and search algorithm of our dispatchers to the use of job duration predictions to obtain high QoS levels in terms of job waiting times and slowdown.

We conduct a simulation study on workload traces collected from HPC systems containing large numbers of short jobs. We use predictions with different accuracy, underestimation and overestimation rates on the dataset. Our results demonstrate that with our approach, the CP-based dispatchers can: (i) significantly reduce the time required to generate dispatching decisions; and (ii) benefit from good job duration predictions and considerably decrease the waiting times and the slowdown of the jobs, especially for workloads dominated by short to medium jobs.

In Sections 5.1 and 5.2, we describe our approach. In Sections 5.3 and 5.4, we detail our experimental study and present our results. We summarize our contribution in Section 5.5.

This chapter corresponds mainly to the publication [50].

5.1 Resiliency to heavy workloads

We revisit the model and search control of the CP-based dispatchers in an effort to make the dispatchers resilient to heavy workloads and applicable to on-line dispatching by reducing the model size and the time to look for a good quality solution.

At a dispatching time t, `PCP` searches for a solution for all the jobs in Q which can be very time consuming when many jobs are waiting. While this problem is tackled in `HCP` by searching for a solution for the jobs in $\bar{Q}$ and postponing the remaining jobs in $Q \setminus \bar{Q}$ to the end of the makespan, there raises another issue: when many jobs are postponed in the same way, they are likely to overlap and create excess demand for the system resources at a given time in the schedule. It may therefore not be possible to find a feasible solution that satisfies the resource constraints, consequently the entire Q may be postponed to the next dispatching time $t + 1$. To address this problem, we remove the remaining jobs jobs in $Q \setminus \bar{Q}$ from the model and place them in the queue with their original q_i.

During the typical operation of an HPC system, job submission by users has a stochastic nature and actual runtime durations are known only when jobs terminate. Additionally, at a dispatching time t, only the jobs with $s(\tau_i) = t$ are dispatched. Thus, it is not fruitful to generate a dispatching decision for the entire schedule makespan $[t, eoh]$. We therefore remove from the model all the jobs requiring more amount of resources than available at time t and queue them again with their original q_i. In addition to reducing the model size in terms of decision variables, we also eliminate the unnecessary variables and constraints in the model of a given problem instance. Specifically, for a given resource type r (in a node n), if none of the jobs in the model require it, we remove the corresponding `cumulative` constraint from the model. Moreover, in `PCP`, if we have $rn_i > \sum_{n \in N} p_{i,n}$ for a job i, this means that there is no availability to allocate i in the system resources, we remove i and its corresponding `alternative` constraint from the model, and queue it again with its original q_i. Note that removing jobs from the model and putting them back in the queue does not cause any starvation problem. As we will argue in Section 5.2 and confirm experimentally in Section 5.4, their priority grow with their slowdown and eventually they are all dispatched.

During search for a solution, both solvers of `PCP` and `HCP` use a time limit δ to interrupt the search and return the best solution found. If, no solution is found within the limit, the search is restarted with an increased time limit $2 * \delta$. In the latter case, the dispatchers cannot distinguish an unsatisfiable problem instance from a difficult instance that is not solved yet. This has the consequence of searching for a solution again and again for an instance known to be unsatisfiable. To address this problem, we add the solver state to the search control. Consequently, if the solver proved unsatisfiability, this will be known when the search is interrupted by the time limit, and the subsequent restart will be avoided by placing the jobs in the queue for the next dispatching time. Finally, we avoid a restart if the solution quality did not change after k consecutive restarts.

In the following, we refer to the versions of PCP and HCP whose model and search control are built as described here as $\texttt{PCP}_1$ and $\texttt{HCP}_1$.

5.2 Incorporation of job duration prediction

A straightforward way to incorporate the duration prediction d_i^d of a job i into our dispatchers is to use it for defining the duration $d(\tau_i)$ as $d(\tau_i) = d_i^d$ for the queued jobs and $d(\tau_i) = s(\tau_i) + d_i^d - t$ for the running jobs, without any other changes to the dispatchers. In this section, we argue that this naive use may be ineffective, if not worsen the QoS, thus we adapt the model and search algorithm of both dispatchers to the use of job duration predictions in order to obtain high QoS levels in terms of job waiting times and slowdown.

A duration prediction d_i^d of a job i may be perfectly accurate ($d_i^d = d_i^r$), underestimated ($d_i^d < d_i^r$), or overestimated ($d_i^d > d_i^r$). If a running job i is underestimated, at a certain dispatching time t, we will have $s(\tau_i) + d_i^d < t$ and thus $d(\tau_i) = s(\tau_i) + d_i^d - t < 0$. That is, the duration of a running job will have a negative value even if the job is still running. A negative $d(\tau_i)$ for a running job directly affects the calculation of the makespan $\sum_{\tau_i} d(\tau_i)$ of the queued jobs. With a reduced makespan, it may not be possible to find a schedule and/or allocation for the queued jobs, consequently they may all be postponed to the next dispatching time $t + 1$, worsening the QoS. If instead, a running job is overestimated at t, we will surely have $s(\tau_i) + d_i^d > t$ and $d(\tau_i) > 0$, thus the makespan will not be shorter than necessary.

To *address the problem of duration prediction underestimation*, we extend the duration $d(\tau_i)$ of a running job i which has $d(\tau_i) < 0$ at time t. Specifically, we redefine it as $d(\tau_i) = 1$, assuming that the job i needs at least one more unit of time as of t. This value is necessary and sufficient. It is the minimum value necessary to prevent a feasible problem instance from turning into an unfeasible one, as the makespan will be large enough to fit all the queued jobs in a schedule. To show that it is sufficient, we remind that at t, only the jobs for which the dispatcher decides that $s(\tau_i) = t$ are dispatched (the remaining are queued again). The allocation decision made for such jobs is valid until the next dispatching time $t + 1$ and is not affected by the actual runtime durations of the running jobs even if they are underestimated. By using the minimum possible value for the duration of the underestimated running jobs, we keep the search space size compact. Our initial experiments confirm that higher values of $d(\tau_i)$ make the problem more difficult. In the following, we refer to this version of $\texttt{PCP}_1$ and $\texttt{HCP}_1$ as $\texttt{PCP}_2$ and $\texttt{HCP}_2$.

Even if the job duration prediction is accurate, resulting in $d_i^d \sim d_i^r$ for all

Enhancement	PCP$_1$	HCP$_1$	PCP$_2$	HCP$_2$	PCP$_3$	HCP$_3$
Reduced model size, improved search control.	✓	✓	✓	✓	✓	✓
Addressing duration prediction underestimation.			✓	✓	✓	✓
Job durations in the obj. function and search.					✓	✓

Table 5.1: Dispatcher versions.

jobs, the dispatchers may still not be able to exploit them fruitfully for targeting low job waiting time $s(\tau_i) - q_i$ and slowdown $(s(\tau_i) - q_i + d_i^r)/d_i^r$. As we saw in Section 2.3, both dispatchers assign a priority to the jobs that should not wait long. Then the jobs with higher priority are forced to be scheduled first via the objective function, as well as in the custom search of the scheduling problem and in the heuristic search of the allocation problem of the `HCP` dispatcher. However, job duration $d(\tau_i)$ is ignored in the priority. It is used only as a tie breaker among the jobs having the same priority during the search of the scheduling and the allocation problems of `HCP`. The priority instead focuses on a relation between the current waiting time $t - q_i$ of the job i and its expected waiting time ewt_i. The problem is that ewt_i is not a job specific feature that can be decided on-line at the time of dispatching. It is a feature of the queue where the job is submitted and is calculated offline. Such a value may not be informative on the current job submission status so as to generate a dispatching decision of high quality.

We tackle this limitation by involving *job durations in the objective function and in the search* of the scheduling and allocation, via the use of job slowdown as job priority. Thus, the new objective function and the priority of a job i at a dispatching time t become $\sum_{\tau_i} \frac{s(\tau_i)-q_i+d(\tau_i)}{d(\tau_i)}$ and $(t - q_i + d(\tau_i))/d(\tau_i)$, respectively. This is the normalization of the job waiting time, which has a higher value for jobs waiting more than their duration than for jobs waiting less than their duration. We foresee the following benefits. First, since it gives priority to short jobs, the dispatcher will aim at lowering both the total job waiting times and the total job slowdown, as required by modern HPC applications. Our experimental results in Section 5.4 show that by giving priority to short jobs, we never penalize the medium and long jobs. Second, it prioritizes the jobs based on a job specific feature $d(\tau_i)$ which can be calculated on-line and which can reflect better the current job submission status. Finally, integrating $d(\tau_i)$ in the objective function and search of the dispatchers paves the way to exploit job duration predictions.

In the following, we refer to the versions of `PCP`$_2$ and `HCP`$_2$ whose model and search algorithms are adapted as described here as `PCP`$_3$ and `HCP`$_3$. Table 5.1 summarizes all the dispatcher versions. We note that, similar to `HCP`, the `HCP`$_3$ dispatcher uses the job priorities in the custom search of the scheduling problem and in the heuristic search of the allocation problem, and calculates the priorities once statically at the dispatching time t before search starts. Our initial experiments revealed that updating them dynamically during search is not beneficial. As we described in Section 2.3, the search of `PCP` relies on the default search of the underlying solver and does not exploit priorities. We observed in our initial experiments that the custom search of the scheduling model in `HCP` is valuable also for `PCP` to solve the entire scheduling and allocation problem, hence we adopt that kind of search and exploit priorities also in $\texttt{PCP}_3$.

5.3 Experimental study

To evaluate the significance of our approach, we conducted an experimental study, by simulating on-line job submission to an HPC system.

HPC systems and their workload dataset Our study is based on two workload traces collected from the Eurora and the Gaia system 2.1.4.

We remind some information about the workload datasets used here. Eurora is an HPC system with a heterogeneous architecture composed of 64 nodes. The workload dataset consists of logs for over 400,000, and it is dominated by short jobs (under 1 hour), making up 93.14% of all jobs, while the remaining 6.10% are medium jobs (between 1 and 5 hours) and 0.75% are long jobs (over 5 hours). Whereas Gaia is an HPC system is a system with more nodes than Eurora, reaching 151 nodes, although it include some resource heterogenity, the workload dataset format does not allow to distinguish different request than core or RAM. It almost has 52,000 jobs, also leading short jobs with 66.5% of the total, and more medium and long jobs than in Eurora, with 17.7% and 15.8%, respectively.

Job duration prediction To derive job durations, we used three prediction methods with varying accuracy levels, and underestimation and overestimation rates: (i) the wall-time approach, with a high overestimation rate and low underestimation rate, (ii) a data-driven prediction heuristic (Section 4.1), and is more accurate than the wall-time approach, with considerable underestimation and overestimation rates, and as a baseline (iii) the actual runtime (real) durations which provide the most accurate prediction with zero underestimation and overestimation rates.

The next results for the Eurora workload dataset are from Section 4.1, although we provide extra analysis to it. The mean absolute error (MAE) of the heuristic and the wall-time approach with respect to the real duration were shown to be 40 mins and 225 mins, respectively. The heuristic prediction shows thus an improvement of 82% over the wall-time approach. The time locality of job durations for individual users is not specific to the Eurora workload. We observed it also in other workload traces, such as Gaia, which was collected by the OAR Batch Scheduler.[43] Applying the data-driven heuristic to this work-

[43]http://oar.imag.fr/

Workload	Prediction method	Under. rate	Over. rate
Eurora	wall-time	3.6%	96.3%
Eurora	heuristic	25.8%	53.7%
Gaia	wall-time	3.0%	97.0%
Gaia	heuristic	40.63%	58.8%

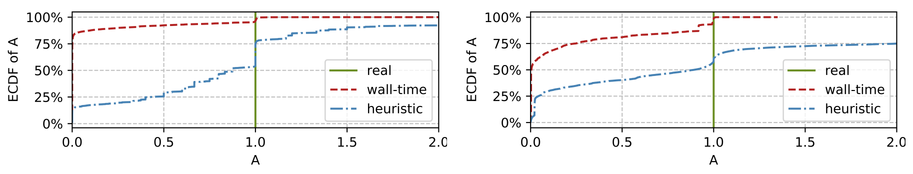
Table 5.2: Underestimation and overestimation rates of the wall-time approach and the prediction heuristic on the Eurora and Gaia workload datasets.

load, we obtained the MAE of 423 mins, while the wall-time approach achieves a MAE of 2,913 mins. Although the dataset has been pre-processed the accuracy of the prediction is improved. The pre-processing removed attributes like job names and replaced them with a number representing the application that the job runs. We note this may occur only in a simulation scenario, where not all workload data is available, whereas in a real system can be accessible.

Table 5.2 gives the underestimation and overestimation rates of both prediction methods on each dataset. The heuristic prediction cuts down significantly the overestimation with respect to the wall-time approach on both datasets, especially on the Eurora dataset. On the other hand, the heuristic introduces considerable underestimation in both datasets, especially on the Gaia one. We note that on Eurora, the heuristic prediction increase the perfect prediction from 0.1% to 20% with respect to wall-time. Where as on Gaia from 0% to 0.6%. We remark the low improvement of the prediction accuracy of the heuristic approach on the Gaia dataset, with respect to Eurora, is due to its less availability of the job attributes of the workload dataset. The Gaia dataset has been pre-processed and therefore attributes like job names are not available, instead, a job is identified by the number of the application it runs.

Next, we evaluate the intensity of the underestimation and overestimation. In Figure 5.1, we show the empirical cumulative distribution function (ECDF) of the prediction accuracy $A = d_i^r/d(\tau_i)$, the ratio between the real and the predicted duration of a job, of all the three methods. The empirical ECDF shows the proportion of scores that are less than or equal to each score of A on Eurora and Gaia datasets. When $A = 1$, the duration $d(\tau_i)$ matches the real duration d_i^r. We have underestimation when $A > 1$, overestimation when $A < 1$. In theory, we should not have underestimation with the wall-time approach because in a real system a job is killed if it takes longer than its d_i. However, a system requires extra time after a job is killed or completed to bring the used resources on-line again and this extra time is reflected to the dataset. Therefore, in some cases we have $A > 1$ in Figures 5.1. In the Eurora dataset, we have $0.75 \leq A \leq 1.25$ for about 50% of the workload with the heuristic, and for less than 10% with the wall-time approach. Whereas in the Gaia dataset is only 35%.

Experimental setup We used the open-source discrete event simulator AccaSim (Chapter 3) to simulate the HPC systems with their workload datasets.



(a) Eurora dataset. (b) Gaia dataset.

Figure 5.1: The distribution of the accuracy of the three prediction methods applied to Eurora and Gaia datasets.

Each job submission is simulated by using its available data, for instance, the owner, the requested resources, and the real duration, the execution command or the name of the application executed. AccaSim uses the real duration extracted from the workload dataset to simulate the job execution during its entire duration despite the presence of underestimation of the job duration. Therefore job duration prediction errors do not affect the running time of the jobs with respect to the real workload data. The dispatchers under study are implemented using the AccaSim directives to allow them to generate the dispatching decisions during the system simulation.

With the heuristic prediction, as opposed to calculating the predictions offline as in Section 4.1, we calculate them on-line during the simulation and update the knowledge base upon job terminations. The accuracy of the heuristic thus depends on the generated dispatching decisions. As a CP modelling and solving toolkit, we used Google OR-Tools[44] version 6.7 and ported it to Python 3.6 to implement the dispatchers in AccaSim.

to small values to keep the dispatcher overhead low. We keep $m = 100$ as in `HCP`. Both dispatchers need in some of their versions the estimated waiting time $ewt_Q = \frac{1}{|Q|} \sum_{i \in Q} s_i - q_i$ of each queue Q in the system. These values were calculated for the Eurora workload in [9, 18] and reused here. We calculated the values of the Gaia by analyzing the workload data which was collected when Gaia was using the OAR Batch Scheduler.[45]

All experiments were performed on a dedicated server with a 16-core Intel Xeon CPU and 8GB of RAM, running Linux Ubuntu 16.04. The source code of the CP-based dispatchers is available at `https://git.io/fjia1`.

5.4 Experimental results

In this section, we report our experimental results. While the best and the final versions of the dispatchers are `PCP`$_3$ and `HCP`$_3$, all the previous versions (`PCP`, `PCP`$_1$, `PCP`$_2$, `HCP`, `HCP`$_1$, `HCP`$_2$) appear in the experiments in order to evaluate each of our contributions. To refer to a dispatcher using a certain job duration

[44]`https://developers.google.com/optimization/`
[45]http://oar.imag.fr/

prediction method, we append `-W`, `-D` or `-R` to the name of the dispatcher for the Wall-time approach, the Data-driven heuristic and the Real duration, resp.

5.4.1 Dispatcher performance and problem size

We first assess the impact of reducing the model size and improving the search control of the dispatchers for resiliency to heavy workloads. Following the original dispatchers `PCP` and `HCP`, we use the wall-time approach in $\texttt{PCP}_1$ and $\texttt{HCP}_1$ for job duration prediction, and compare the performance of and the problem size in `PCP` and $\texttt{PCP}_1\texttt{-W}$, as well as `HCP` and $\texttt{HCP}_1\texttt{-W}$.

Eurora

We report in Table 5.3 the mean CPU time spent in generating a dispatching decision over all dispatcher invocations, including the time for modeling the dispatching problem instance and searching for a solution. We also report the total simulation time from the first job submission until the last job completion, and the average problem size: number of intervals, number of requested resources, number of available resources.

We cannot report the results of `PCP` since it crashes due to a memory allocation problem before the completion of the entire workload, demonstrating that it this dispatcher is not resilient to heavy workloads.

Dispatcher	PCP	$\texttt{PCP}_1\texttt{-W}$	$\texttt{PCP}_3\texttt{-R}$	$\texttt{PCP}_3\texttt{-D}$	HCP	$\texttt{HCP}_1\texttt{-W}$	$\texttt{HCP}_3\texttt{-R}$	$\texttt{HCP}_3\texttt{-D}$
Avg. disp. time [ms]	∞	743	692	701	1,014	703	523	575
Total pred. time [s]	-	-	-	289	-	-	-	308
Total sim. time [s]	∞	262,436	261,985	262,764	374,788	245,663	201,223	215,814
Avg. # of intervals	-	145	94	115	379	100	51	63
Avg. # of req. res.	-	853	142	584	6,267	1,292	258	571
Avg. # of avl. res.	-	1,476	1,471	1,473	1,487	1,477	1,473	1,474

Table 5.3: Times and problem sizes.

We therefore underline the improvement reached by the $\texttt{PCP}_1\texttt{-W}$ dispatcher which is now able to process the workload. Compared to `HCP`, the $\texttt{HCP}_1\texttt{-W}$ dispatcher reduces the total time by around 34% and reduces the problem size and time required for dispatching significantly.

These results demonstrate that our approach has significantly better performance, making the dispatchers applicable to heavy workloads and paving the way to the use of CP-based dispatchers for HPC on-line dispatching.

Gaia

Similarly to the Eurora results, we report in Table 5.4 the mean CPU time spent in generating a dispatching decision over all dispatcher invocations, and the total simulation time from the first job submission until the last job completion, and the average problem size.

Dispatcher	PCP	$\texttt{PCP}_1\texttt{-W}$	$\texttt{PCP}_3\texttt{-R}$	$\texttt{PCP}_3\texttt{-D}$	HCP	$\texttt{HCP}_1\texttt{-W}$	$\texttt{HCP}_3\texttt{-R}$	$\texttt{HCP}_3\texttt{-D}$
Avg. disp. time [ms]	∞	78	99	89	103	82	85	84
Total pred. time [s]	-	-	-	102	-	-	-	110
Total sim. time [s]	∞	6,380	8,155	7,445	8,561	6,781	7,016	7,090
Avg. # of intervals	-	260	260	261	2	2	2	2
Avg. # of req. res.	-	2	2	2	13	13	13	13
Avg. # of avl. res.	-	1,091	1,092	1,092	1,089	1,090	1,089	1,092

Table 5.4: Times and problem sizes.

These results strengthen our results of Eurora experiments, confirming that rebuilding the model and search control of both dispatchers, has significantly better performance, significantly cuts down the time spent in generating dispatching decisions and the problem size, making thus the dispatchers applicable to heavy workloads and paving the way to the use of CP-based dispatchers for HPC on-line dispatching.

5.4.2 Quality of the dispatching decisions

Next, we evaluate the value of adapting the model and search algorithm of the dispatchers to the use of job duration predictions by comparing the quality of the decisions made by PCP, $\texttt{PCP}_1$, $\texttt{PCP}_2$, $\texttt{PCP}_3$, as well as by HCP, $\texttt{HCP}_1$, $\texttt{HCP}_2$, $\texttt{HCP}_3$. Since we are aiming at reducing both the slowdown and waiting time of jobs, we consider both of these metrics to evaluate the quality of the dispatching decisions. We calculate the waiting time and slowdown of a job after it is dispatched and completed, respectively.

We first study the effectiveness of PCP, $\texttt{PCP}_1$, $\texttt{PCP}_2$, and $\texttt{PCP}_3$ with each job duration prediction method. Then, we analyze HCP, $\texttt{HCP}_1$, $\texttt{HCP}_2$, and $\texttt{HCP}_3$. We show the results of $\texttt{PCP}_2$ and $\texttt{HCP}_2$ only in conjunction with the data-driven heuristic. This is because on our workload datasets the heuristic has a considerable underestimation rate while the other prediction methods have negligible or no underestimation, so the behaviour was very similar to $\texttt{PCP}_1$ and $\texttt{HCP}_1$.

Eurora

We also compare the various dispatchers with the performance of PBS in the original system, by calculating the slowdown and waiting time from the workload data.

PCP results Figure 5.2 shows the slowdown and waiting times obtained by various versions of the dispatchers, compared to PBS. PCP is missing from the plot due to the fact that it is not able to process the workload, hence we consider $\texttt{PCP}_1\texttt{-W}$ as a baseline, which is the enhancement most similar to the original algorithm. Additionally, we do not report the results of $\texttt{PCP}_1\texttt{-D}$ on Eurora because the simulation was too heavy and did not terminate in more than two weeks, so we interrupted it. We believe the long simulation time is due to the fact that $\texttt{PCP}_1\texttt{-D}$ does not deal with underestimation, so it tends to use the maximum time limit for the instances in which jobs are underestimated, generating long queues.

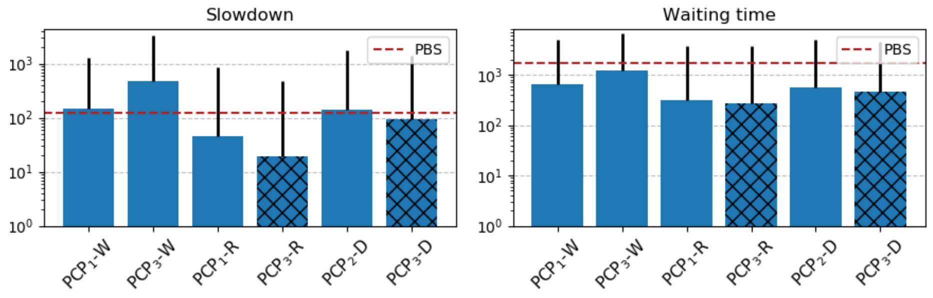


Figure 5.2: Average and error bars showing one standard deviation of slowdown and waiting times [s] using the `PCP` dispatchers.

A first observation is that, our best dispatcher coupled with the best duration predictor (`PCP`$_3$`-R`) and the heuristic predictor (`PCP`$_3$`-D`) always outperform PBS. `PCP`$_3$`-W` has lower performance compared to `PCP`$_1$`-W`. This is probably because the wall-time approach has a high overestimation rate, which is not beneficial when the dispatcher involves job durations in dispatching decisions. However, if we look at the dispatchers using real durations, we observe a significant increase in performance compared to `PCP`$_1$`-W` but also when moving from `PCP`$_1$`-R` to `PCP`$_3$`-R`. The reduction in the slowdown and waiting time from `PCP`$_1$`-R` to `PCP`$_3$`-R` reach up to 58% and 13%. This is due to the accuracy of the prediction method which does not present any underestimation nor overestimation. This proves that our approach is essential when a good quality prediction is available.

On a more realistic prediction, the results confirm that great care needs to be taken when integrating predictions. A straightforward integration of the predictions in previous algorithms is not helpful at all: `PCP`$_1$`-D` takes too long. By handling underestimation as in `PCP`$_2$`-D`, we are able to improve the results compared to `PCP`$_1$`-W`. Further improvement is observed when moving to `PCP`$_3$`-D`, demonstrating again the benefits of including predictions, albeit imperfect, into the model and search algorithm. Specifically, we observe 37% and 29% reduction in the average slowdown and the average waiting time.

HCP results Figure 5.3 shows the performance of `HCP`, `HCP`$_1$, `HCP`$_2$ and `HCP`$_3$ compared to PBS. Unlike the `PCP` case, here the original dispatcher `HCP` is able to process the entire workload so we can compare our results directly with the state-of-the-art method, besides PBS. We observe that in general, if we include predictions with good accuracy and take into account also the underestimation problem, our algorithms can improve the quality of the dispatching decisions significantly (see `HCP`$_3$`-D` and `HCP`$_3$`-R` compared to `HCP` and PBS).

In more detail, we observe that simply moving from `HCP` to `HCP`$_1$`-W`, with an approach aimed at reducing the CPU time for dispatching, we also improve the quality of the solutions. `HCP`$_3$`-W` does not improve `HCP`$_1$`-W`, since the accuracy

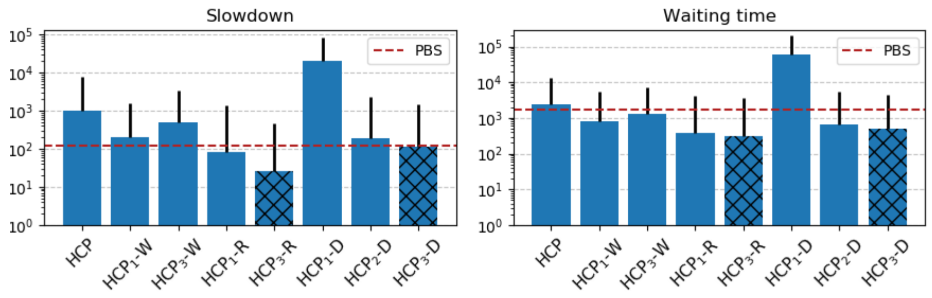


Figure 5.3: Average and error bars showing one standard deviation of slowdown and waiting times [s] using the HCP dispatchers.

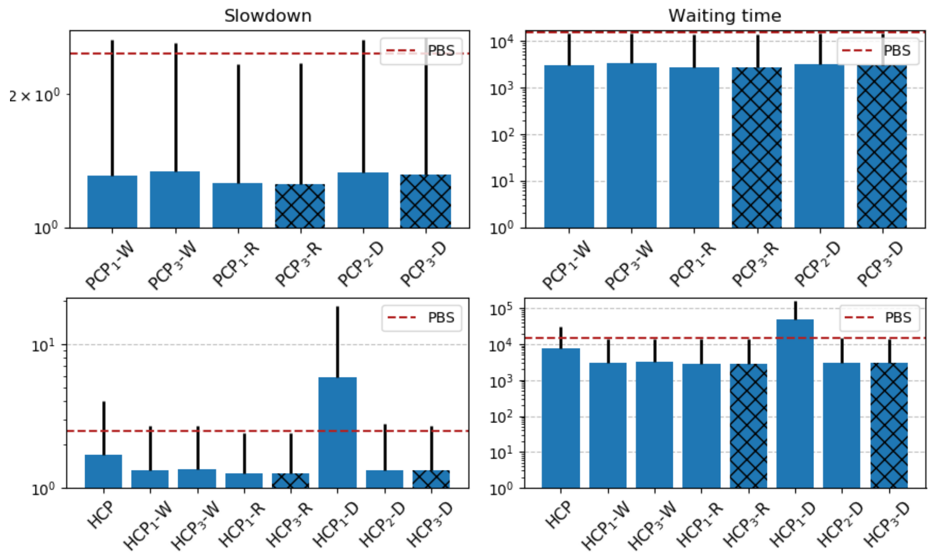


Figure 5.4: Average and error bars showing one standard deviation of slowdown and waiting times [s] on medium and long jobs using all the dispatchers on the Eurora workload.

of predictions using wall-time is rather low. We observe the most significant improvements over HCP with $\mathtt{HCP}_3\mathtt{-R}$, proving again the importance of our approach when a good quality prediction is available. The decreased performance of $\mathtt{HCP}_1\mathtt{-D}$ compared to all other algorithms confirms again that naively including predictions can be detrimental. The gains obtained by $\mathtt{HCP}_2\mathtt{-D}$ with respect to $\mathtt{HCP}_1\mathtt{-D}$ support again the need of dealing with underestimated jobs. We note that, while $\mathtt{HCP}_1\mathtt{-D}$ performs worse than the original HCP, $\mathtt{HCP}_2\mathtt{-D}$ becomes better than HCP and $\mathtt{HCP}_3\mathtt{-D}$ further improves $\mathtt{HCP}_2\mathtt{-D}$, demonstrating again the benefits of including predictions, albeit imperfect, into the model and search algorithm.

Gaia

We also compare the same dispatchers by calculating the slowdown and waiting time. We cannot calculate the performance of OAR in the original system because the workload dataset is not in line with the simulation (initial submissions show a delayed start).

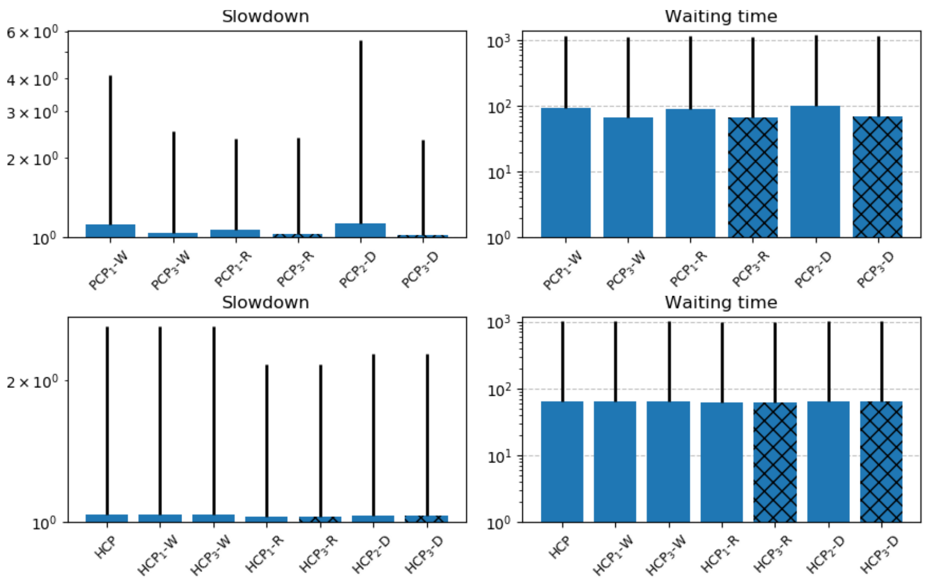


Figure 5.5: Average and error bars showing one standard deviation of slowdown and waiting times [s] on Gaia using the `PCP` and `HCP` dispatchers.

Figure 5.5 shows the results of `PCP` and `HCP` dispatchers applied to the Gaia system. The results are very similar regarding the Eurora ones, although on a lower scale. This scale is due to the size of the workload, and probably to the fewer possibilities to optimize the decisions because, during less time, the system is fully occupied. Despite this, the results showed in a different system that our approach is also valid.

5.5 Summary

The state-of-the-art CP-based dispatchers [9, 18] are unable to satisfy the challenges of on-line dispatching and to take advantage of job duration predictions, which impede their adoption in HPC systems. We have introduced a class of novel CP-based dispatchers by building on [9, 18] and redesigning their main components. We made them resilient to heavy workloads and applicable to on-line dispatching, as well as adapted them to the use of job duration predictions to obtain high QoS levels in terms of job waiting times and slowdown. We evaluated the significance of our approach on two workload traces collected from two HPC systems, Eurora and Gaia systems, using predictions with different accuracy and underestimation and overestimation rates on the datasets.

We can conclude that suitable incorporation of job duration predictions in PCP and HCP, such as $\texttt{PCP}_3$ and $\texttt{HCP}_3$, can lead to significantly higher levels of QoS in terms of job waiting times and slowdown, especially for workloads dominated by short jobs. To benefit from this potential, durations should rely on predictions with acceptable levels of accuracy, going beyond the standard walltime approach. The quality of the decisions generated by $\texttt{PCP}_1\texttt{-W}$ and $\texttt{HCP}_1\texttt{-W}$ is much worse than $\texttt{PCP}_3\texttt{-R}$ and $\texttt{HCP}_3\texttt{-R}$. On the other hand, $\texttt{PCP}_3\texttt{-D}$ and $\texttt{HCP}_3\texttt{-D}$ offer valid alternatives to $\texttt{PCP}_1\texttt{-W}$ and $\texttt{HCP}_1\texttt{-W}$ with further reductions in problem size (as reported in Table 5.3 and Table 5.4) and with QoS measures closer to those of $\texttt{PCP}_3\texttt{-R}$ and $\texttt{HCP}_3\texttt{-R}$. The gains offered by $\texttt{PCP}_3\texttt{-D}$ and $\texttt{HCP}_3\texttt{-D}$ are prominent especially on the Eurora dataset, while on the Gaia dataset the major advantage is in reducing the waiting times. Yet, there is still room for improvement in the QoS of $\texttt{PCP}_3\texttt{-D}$ and $\texttt{HCP}_3\texttt{-D}$, as can be witnessed by comparing them with $\texttt{PCP}_1\texttt{-R}$ and $\texttt{HCP}_1\texttt{-R}$ in all the figures. It seems thus necessary to reduce the underestimation and overestimation of the prediction as much as possible. At the same time, as shown in Table 5.3, $\texttt{PCP}_3\texttt{-D}$ and $\texttt{HCP}_3\texttt{-D}$ decrease in most of the cases the average time incurred to generate a dispatching decision with respect to $\texttt{PCP}_1\texttt{-W}$ and $\texttt{HCP}_1\texttt{-W}$. Table 5.3 shows also the time cost of this gain. While $\texttt{PCP}_3\texttt{-D}$ and $\texttt{HCP}_3\texttt{-D}$ come each with a cost of prediction, the total simulation times of $\texttt{PCP}_3\texttt{-D}$ and $\texttt{PCP}_1\texttt{-W}$ are similar, and $\texttt{HCP}_3\texttt{-D}$ reduces notably the time with respect to $\texttt{HCP}_1\texttt{-W}$. The fact that the new dispatchers give priority to short jobs does not penalize the medium and long jobs, as can be witnessed in Figures 5.4. Finally, our approach does not affect the system utilization. We did not observe any major differences between the various dispatchers probably because all the dispatchers are using the best-fit allocation strategy.

Chapter 6

Towards system size-independent CP-based job dispatchers for HPC systems

So far, we tested our improvements regarding the scalability to heavy workloads in two systems that are far from most of HPC systems in the TOP500 list that, in general, have more than 500 computing nodes. Eurora and Gaia, in Figure 6.1, have 64 and 151 computing nodes, respectively, so, how our `PCP`$_3$ and `HCP`$_3$ dispatchers may scale to bigger systems?

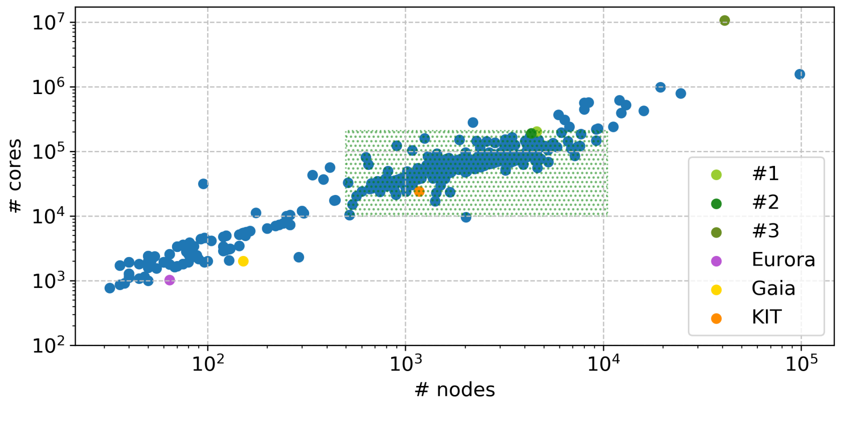


Figure 6.1: Size of Eurora, Gaia and KIT, and TOP500 HPC systems.

`HCP`$_3$ may scale well on big systems because it is independent of the size of the system, and only uses $\bar{Q}$ decision variables to model the scheduling problem. Despite this, the dispatching decision may be weak given its uncoupled approach. For this reason, we believe CP-based dispatchers that model the scheduling and allocation problems together in one model, may incorporate additional restrictions and provide better dispatching decisions for current and future HPC systems.

In the previous chapter, $\texttt{PCP}_3$ showed outstanding results in small systems and mediums systems such as Eurora and Gaia. However, $\texttt{PCP}_3$ may not scale well on big systems due to how its decision variables are modeled. The number of decision variables depends on the number of queued jobs and all of their possible allocations on each computing node, thus the $\texttt{PCP}_3$ model is highly dependant on the system size. In detail, for a given instance of the on-line job dispatching problem, $\texttt{PCP}_3$ uses $|\bar{Q}|$ decision variables to model the scheduling problem and $\sum_{i\in\bar{Q}}\sum_{n\in N} p_{i,n}$ decision variables to model the allocation problem, where $p_{i,n} = min(rn_i, \min_{r\in R}\lfloor\frac{cap_{n,r}}{req_{i,r}/rn_i}\rfloor)$, and, rn_i and $req_{i,r}$ are the number of requested nodes and the number of requested r resources by i, respectively; $cap_{n,r}$ is the capacity of the resource r in n. To illustrate the number of decision variables employed in the $\texttt{PCP}_3$ model, let us consider a simple serial job i, i.e. $rn_i = 1$, so the decision variables to model i will be $1 + |N|$, with N as the set of computing nodes. If the system is small, such as Eurora, the model will use 65 decision variables in the worst case, instead, if the job requests MIC or GPU resources, it will be 33 decision variables. Thus, this model in a bigger system will use more decision variables, even more, if jobs are highly parallel ($rn_i > 2$) and still more if many jobs are in $|\bar{Q}|$. Thus, this model may not be suitable on a system with a higher number of nodes due to the increment of decision variables and consequently requiring more time to generate a dispatching decision.

Therefore, in this chapter, we propose new dispatchers to deal with the online job dispatching problem, which are composed of fewer decision variables than the utilized in $\texttt{PCP}_3$ and do not depend proportionally with the size, in terms of the number of nodes, of the system and the allocation possibilities of the job units. We believe such models due to this independence may scale better to big systems.

This chapter is structured as follows. In Section 6.1, we present a new job dispatcher, named $\texttt{PCP}_4$, which models the on-line job dispatching using $|\bar{Q}| + \sum_{i\in\bar{Q}} |N|$ decision variables in its model. This model is inspired in the $\texttt{PCP}_3$ model, however, it changes the type of allocation variables from intervals to integer variables. Instead, in Section 6.2, we present another job dispatcher, named $\texttt{PCP}_5$, which is a new model and uses $|\bar{Q}|+\sum_{i\in\bar{Q}} rn_i{*}|R|$ decision variables to model the same problem. Both models put more emphasis in the allocation problem, leaving the scheduling problem as a subordinated one. Therefore, advanced allocation decisions can be implemented, as the allocation strategies proposed in [93] for heterogeneous resource architectures. Next, in Section 6.3, we describe a possible search for both models. Similar to $\texttt{PCP}_3$, this search puts emphasis to short jobs via slowdown prioritization. In the remaining sections, we describe our experiments and analyze the results them in Section 6.4 and 6.5, respectively. Finally, we summarize our contribution in Section 6.6.

6.1 $\texttt{PCP}_4$ model: Variable number of job units assignments

We model every job i as a Non-Optional Fixed Interval Variable τ_i, so each job in the model has a fixed duration and must be scheduled at $s(\tau_i) \geq t$. A problem instance at time t may include running jobs, for which the start time $s(\tau_i) < t$ was already decided, so only start times for queued jobs in $\bar{Q}$ are unknown with $D(s(\tau_i)) = [0, \dots, eoh]$. During the execution of a job i, $req_{i,r}$ resources of types r are consumed by rn_i job units (number of requested nodes).

Identically to $\texttt{PCP}_3$, $\texttt{PCP}_4$ incorporates the duration prediction d_i^d of a job i by defining the duration $d(\tau_i)$ as $d(\tau_i) = d_i^d$ for the queued jobs and $d(\tau_i) = \texttt{max}(s(\tau_i) + d_i^d - t, 1)$ for the running jobs.

A job i may be allocated to different nodes depending on rn_i and the number of times the job units $(req_{i,r}/rn_i)$ can fit on the system nodes N. We model the allocation variables using *integer variables* for each job i, for each node $n \in N$, as $a_{i,n}$ and whose domain is $D(a_{i,n}) = [0, p_{i,n}]$ for queued jobs, where $p_{i,n} = min(rn_i, min_{r \in R}\lfloor cap_{n,r}/req_{i,r} \rfloor)$. Instead, we use the actual allocation for running jobs, which are constant.

We define a set of constraints to ensure a feasible scheduling and allocation decision. First, we post a constraint for each queued job i to limit the number of allocated resources to the nodes:

$$\sum_{n \in N} a_{i,n} = rn_i$$

We use a similar idea from $\texttt{HCP}$ model to model the capacity constraints, i.e. we use a relaxed version of the problem and on top of it, we define the corresponding $\texttt{cumulative}$ constraint for each r resource type. In addition, we define a mirrored version (inverting the axes) of the previous constraint. In detail, we post the first $\texttt{cumulative}$ constraint for each $r \in R$, which is defined as

$$\texttt{cumulative}(\tau_i, req_{i,n} * req_{i,r}, cap_r)$$

Instead, the mirrored version corresponds to

$$\texttt{cumulative}(\tau_i^{'}, d(\tau_i^{'}), eoh)$$

where $\tau_i^{'} = [0, cap_r]$ and $d(\tau_i^{'}) = req_{i,n} * req_{i,r}$.

We also use energetic reasoning to detect anticipatedly infeasible states. The energy is defined as the duration multiplied by a resource quantity. This gives an aggregate amount comparable to a surface, avoiding the specification of other parameters. For each pair of jobs if at least one of them is a queue job, we post a disjunctive constraint if the sum of both energies exceeds the energy of the system for a given r. The energy of a job i is evaluated as $d(\tau_i) * \sum_{n \in N} rn_i * req_{i,r}$, and the energy of the system is calculated as $max(\tau_i, \tau_j) * cap_r$ with $i \neq j$.

At this point, we only ensured the scheduled jobs will not exceed the capacity of the system at each time on the *makespan*. In order to ensure feasible job allocations, we use the Non-Overlapping Boxes constraint [11], also known as `diffn`, for all $n \in N$ and for all $r \in R$. We consider both running jobs consuming r and queued jobs demanding r and can be allocated on n. In this constraint, a box represents a job unit running in the system, consuming $a_{i,n} * req_{i,r}/rn_i$ resources on n during its execution, and it is defined as:

$$\begin{aligned}
x_i &= s(\tau_i) && \text{origin of the box (x-axis)}\\
dx_i &= d(\tau_i) && \text{box width}\\
y_i &= [0, cap_{n,r}] && \text{origin of the box (y-axis)}\\
dy_i &= a_{i,n} * req_{i,r}/rn_i && \text{box height}
\end{aligned}$$

Figure 6.2 presents a graphical interpretation of a box. We remark that for this model the assignment of y_i is irrelevant.

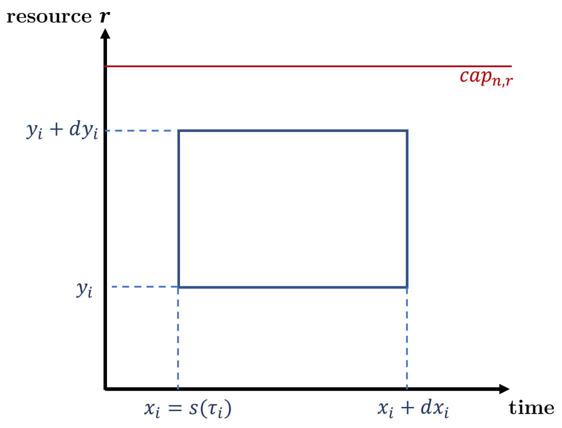


Figure 6.2: Graphical interpretation of a box.

6.2 PCP$_5$ model: Variable job units positions

Similarly to PCP$_4$, we model every job i, running and queued job in $\bar{Q}$, using a Non-Optional Fixed Interval Variable τ_i. This model is principally based on the usage of the `diffn` constraint, and jobs are represented by their job units. Conversely to job units defined in PCP$_3$ and PCP$_4$where depends on the configuration of each node n, in PCP$_5$ job units are defined differently and are independent of the configuration of nodes. rn_i job units $u_i =< req_i/rn_i >$ of a job i are identical in terms of requested resources, and the number of each elements in the tuple depends on the requested resources. As we described in the previous model, for each box two assignments should be found: the origin of boxes in the x-axis and the height of the box for a given origin in the y-axis. However, in the previous model implies to post a `diffn` constraint for each $r \in R$ in $n \in N$. Instead, in this new model, we place together the same class of resources as a single pool of resources, following the order of the nodes. Figure 6.3 represents the pool of resources definition of the Eurora system as example. We recall its architecture; it is composed of 64 nodes. The first 32 nodes were configured with 16 cores, 2 GPU accelerator cards, and 16 GB of memory each one, instead the other half has 2 MIC accelerator cards instead of the GPU ones.

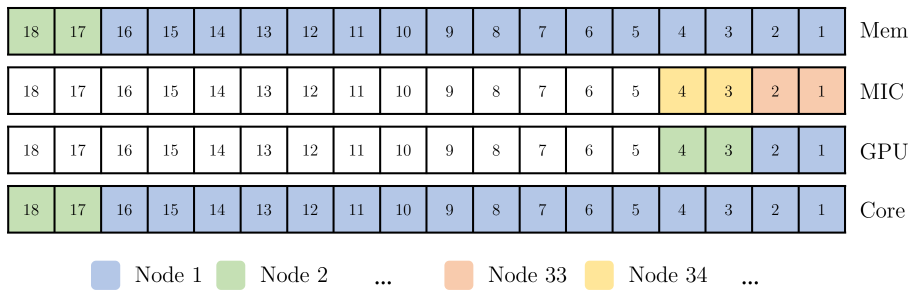


Figure 6.3: Resource modeling of the Eurora system.

Therefore, for each y of each box of job units $u_{i,r}$ of i, from now $y_{i,j,r}$, we need to find an assignment corresponding to the origin of the job unit j for a given resource r. However, this is not straight forward because assignments can be performed on different nodes. To ensure that the assignments of a job unit j of a job i of different resources r are performed on the same node, we define an auxiliary mapping matrix to link the $y_{i,j,r}$ positions to a given node n. Considering the data from Figure 6.3, we define this mapping matrix as detailed in Figure 6.4.

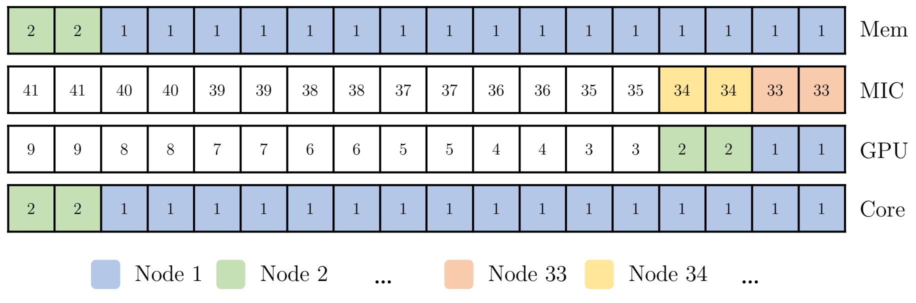


Figure 6.4: Node mapping of the Eurora system.

We employ two classes of constraints using the node mapping matrix *map*. The first one corresponds to allocate a job unit j of a job i on the same node for all the requested resources $req_{i,r} > 0$.

$$\texttt{element}(map_{r_1}, y_{i,j,r_1}) = \texttt{element}(map_{r_2}, y_{i,j,r_2})\ \forall r_1, r_2 \in \hat{R} \mid r_1 \neq r_2$$

where $\hat{R}$ is the subset of requested resources of the job i. The second class of constraints is to ensure that a job unit j will not exceed the position of the allocated node for each r, that is j is constrained to be allocated in a unique node and it will be placed entirely on a single node. Thus the origin $y_{i,j,r}$ of a job unit j of i and the consumption of a resource $dy = req_{i,r}/rn_i$ are constrained to be on the same node:

$$\texttt{element}(map_r, y_{i,j,r}) = \texttt{element}(map_r, y_{i,j,r} + dy_{i,r} - 1)\ \forall r \in \hat{R}$$

Because of job units of a job i cannot overlap among them, their origins should be different, so we force that all $y_{i,r}$ to be different with respect to each resource r.

$$\texttt{alldifferent}([y_{i,r}])\ \forall r \in \hat{R}$$

To ensure feasible job unit allocations, we use the `diffn` constraint for all $r \in R$. We consider both running jobs consuming r and queued jobs demanding r. In this model, we represent boxes as:

$$\begin{aligned} x_i &= s(\tau_i) && \text{origin of the box (x-axis)} \\ dx_i &= d(\tau_i) && \text{box width} \\ y_i &= [0, Tcap_r] && \text{origin of the box (y-axis)} \\ dy_i &= req_{i,r}/rn_i && \text{box height} \end{aligned}$$

where $Tcap_r = \sum_n cap_{n,r}$.

6.3 A possible search for the new models

Both models are composed of two types of decision variables, each one regarding both sub-problems of the dispatching problem, i.e., scheduling and allocation variables. Using the available search methods on OR-Tools, we can combine two sequential searches, as sketched in Figure 6.5, one for the scheduling variables and another for the allocation variables. However, experiments showed this approach is not suitable for online dispatching because the time required for solving the dispatching problems grows with respect to the size of the problem.

Therefore, to tackle efficiently the problem we decided to interleave scheduling and allocation assignments of individual jobs considering jobs' priority. To do so, we dispatch, i.e. schedule and allocate, a job i at a time. This is because,

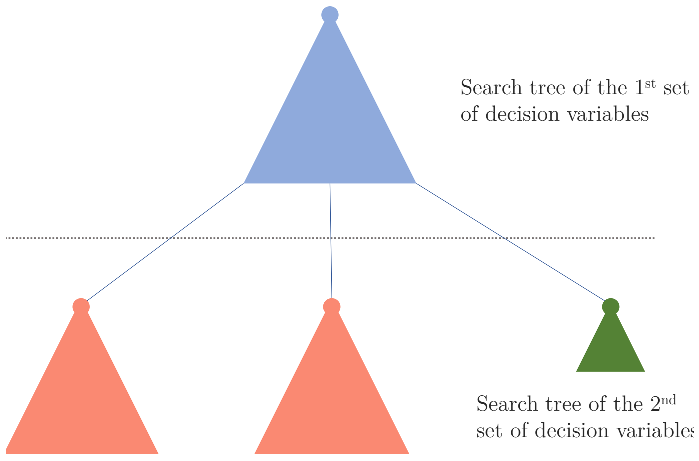

Search tree of the 1st set of decision variables
Search tree of the 2nd set of decision variables

Figure 6.5: Composed searches.

once the start time of a τ_i, is assigned to its earliest starting time, the underlying constraint propagation mechanism of the capacity constraints guarantees a systematic update of the remaining scheduling variables in τ and the allocation variables belonging to τ_i. The capacity constraints guarantee that the start time and during entire execution of τ_i are consistent with the resource requirements of "scheduled" activities, i.e. jobs for which the start time have been fixed. Jobs are selected based on their job priority, i.e. jobs with higher slowdown value are preferred.

This new search algorithm is described in Algorithm 4. NewSearch receives as argument a set of scheduling and allocation variables, S and A respectively. In addition, a variable l to store the last scheduled variable or NULL. Line 1 the UpdateSelectableVariables function returns a tuple of 4 elements. The first element correspond to a flag, which is **true** if all allocations and jobs variables have fixed values, otherwise is **false**. The second and third elements of the tuple correspond to the set of unassigned scheduling and allocation variables in S and A, respectively. The last element in the tuple corresponds to the l variable which is updated to $l := NULL$ if $A'(l) = \emptyset$.

Lines 3-7 perform a scheduling decision. This decision is based on the SelectJob function in line 9, which returns the job with the highest priority and the minimum start time. Jobs are selected based on a priority scheme calculated before the search starts. However, if $S = \emptyset$, the solver will backtrack to the most recent choice point. Choice points are performed in lines 5 and 13, where an assignment of a variable is performed. Later, 11-14 perform an allocation decision for the scheduling variable l. The allocation decision is based on the Best Fit approach, the assignment of allocation variables A belonging to l, will be executed until $A(l) = \emptyset$.

The NewSearch is initialized with $S := \tau$ and $A := X$, i.e. the complete set of scheduling and allocation variables respectively.

Algorithm 4: NewSearch(S,A,l)

```
1  <SOLVED, $S', A', l$ > := UpdateSelectableVariables($S, A, l$)
2  if SOLVED then return
3  if $S \neq \emptyset$ then
4      l := SelectJob(S)
5      D(l) := Min(l)
6      return NewSearch(S,A,$l$)
7  else
8      return Fail
9  end
10 if $l \neq$ NULL then
11     a := MinDomain($A', l$)
12     D(a) := MaxValue(a)
13     return NewSearch($S', A', l$)
14 end
```

6.4 Experimental design

Dispatching evaluation on a big-size system We evaluated if the new and previous CP-based dispatchers, $\texttt{PCP}_3$, $\texttt{PCP}_4$ and $\texttt{PCP}_5$, are able to work in big systems by simulating on-line job submission to an HPC system. We simulated the on-line job submission to the KIT system. Similar to the experiments in the Chapter 5, we use three prediction methods with different accuracy levels, i.e. underestimation and overestimation rates: the job's wall-time (`-W`), which can be user estimates or system default wall-times, the last two method [129] (`-L2`), and as a baseline the actual runtime durations (`-R`). We cannot use `-D` as in our experiments in previous chapter because the workload dataset of the KIT system does not contain any mean of job identification (name of the job, application identification, etc.), so we opted to use `-L2`, which is very a simple heuristic which uses the runtimes of the last two jobs to predict the duration of the next job.

Dispatching evaluation on small/medium-size systems Additionally to the the performance analysis of the dispatchers on a big system, we need to evaluate how the new dispatchers behave with respect $\texttt{PCP}_3$ on small and medium systems. From Chapter 5, we already know that $\texttt{PCP}_3$ is a very good alternative for small and medium systems, such as Eurora and Gaia. We skip the experimental study on the Gaia system because its workload dataset is not heavy and the results tend to be very similar. In this second experiment, we compare how the proposed dispatchers behave regarding the same instance using the same objective function during the same time limits. This experiment will allow discovering different conditions for which dispatcher is better for a given instance. Each instance is obtained by the simulation of the entire Eurora workload dataset with $\texttt{PCP}_3$ as dispatcher and using `-R` and `-D` job duration predictors. We skip `-W` because its inaccuracy affects decisions based on the duration of jobs. An instance of the on-line job dispatching problem corresponds to a state of the simulated system, that is, running jobs and their specific allocations and queued jobs. Thus, an instance is generated every time the WMS calls the job dispatcher during the simulation. To evaluate the dispatchers, we compare the value of the objective function and the required time of each instance solved with the proposed dispatchers, i.e. $\texttt{PCP}_4$ and $\texttt{PCP}_5$, with respect to $\texttt{PCP}_3$. All dispatchers under study use attempt to minimize the same objective function, which corresponds to the jobs' slowdown.We consider the time spent in modeling and solving as the required time for generating a dispatching decision.

Workloads used in the experimental study The KIT ForHLR II system has 1,152 thin nodes with 20 cores and 64 GB memory each one; and 21 fat nodes with 48 cores, 4 GPU accelerator cards, and 1 TB memory each one. Its workload dataset consists in almost 115,000 jobs. The KIT workload dataset is freely available on the Parallel Workloads Archive [46], however, as well as other workload datasets available on the same website, sensitive data, such as job names, were removed. The Eurora system consists of 64 nodes, each equipped with 2 8-core GPUs, 16GB of RAM memory, and 2 accelerators cards: GPUs and MICs – half of nodes has GPUs, the other half MICs; its workload dataset consists of over 400,000 jobs. Further details of both systems are available in Section 2.1.4.

Experimental setup The simulation of the entire workloads and single instances were executed using our AccaSim (Chapter 3) simulator. As a CP modelling and solving toolkit, we used a custom version of Google OR-Tools[47] based on its version 7.3 and ported it to Python 3.6. The OR-Tools implementation of the `diffn` constraint does not include the limits of the placement space, instead, it infers the limits using lower and upper bounds of the real placement space, $x_{min} = \texttt{min}(x)$ and $y_{min} = \texttt{min}(y)$; $x_{max} = \texttt{max}(x) + \texttt{max}(dx)$ and $y_{max} = \texttt{max}(y) + \texttt{max}(dy)$ respectively. Due to the lengths and heights of boxes are unbounded, the possibility of a higher propagation is reduced. Therefore, we propose a new version of the `diffn` constraint in the OR-Tools framework which considers the actual limits of the placement space. These limits are $x_{min} = 0$, $x_{max} = eoh$, $y_{min} = 0$, and $y_{max} = cap_{n,r}$. Given the previous limits, we can post $x_i + dx_i \leq eoh$ and $y_i + dy_i \leq cap_{n,r}$ for each box i to ensure it will be placed inside of the placement space, which corresponds to an feasible allocation on each node during a feasible time.

The OR-Tools implementation of the `diffn` also adds redundant `cumulative` constraints with the aim of performing a finer energy based reasoning to do more propagation. However, the value of the capacities corresponds to the upper

[46]`https://www.cse.huji.ac.il/labs/parallel/workload/logs.html`
[47]`https://developers.google.com/optimization/`

bound of the real placement space, so we replace them with the actual placement limits. We also modify the propagator called *FailWhenEnergyIsTooLarg*. This propagator makes the solver fails if the minimum area of a given box plus the area of its neighbors is greater than the area of a bounding box that necessarily contains all these boxes. The minimum area and bounding box is computed incrementally, adding the neighbor values once at the time. The bounding box is calculated using the upper bounds of the boxes, so this propagator may not fail in some cases. Therefore, we use the actual limits to calculate the bounding box.

All experiments were ran on a CentOS machine equipped with Intel Xeon CPU E5-2640 Processor and 15GB of RAM.

6.5 Experimental results

6.5.1 On-line job dispatching on a big system: The KIT system

As we introduced this chapter, $\texttt{PCP}_3$ is not able to deal with big systems. Indeed, $\texttt{PCP}_3$ stopped dispatching jobs after a while and started to queue incrementally without dispatching a job although the system was empty, i.e. with all the resources of the system available. Looking at the traces of the simulation, $\texttt{PCP}_3$ is not able to handle certain jobs within the time limit, blocking later dispatching decisions. Therefore, we cannot compare either the quality or the spent time by generating dispatching decisions. The jobs that $\texttt{PCP}_3$ cannot deal with are mainly highly parallel jobs, that is job that requests to run on many nodes, setting a very complex allocation scenario. As we introduced, this is a drawback for this dispatcher which relies on the number of nodes of the system and the frequency of its job unit can fit in a node.

Dispatcher	$\texttt{PCP}_3$-W	$\texttt{PCP}_3$-R	$\texttt{PCP}_3$-L2	$\texttt{PCP}_4$-W	$\texttt{PCP}_4$-R	$\texttt{PCP}_4$-L2	$\texttt{PCP}_5$-W	$\texttt{PCP}_5$-R	$\texttt{PCP}_5$-L2
Avg. disp. time [ms]	∞	∞	∞	823	782	807	523	575	537
Total pred. time [s]	-	-	-	-	-	0.5	-	-	0.5
Total sim. time [s]	∞	∞	∞	165,490	156,823	162,564	57,851	108,541	56,196

Table 6.1: Times.

Table 6.1 shows that, in general, both dispatchers can handle big systems efficiently where $\texttt{PCP}_3$ cannot. In detail, $\texttt{PCP}_5$ shows the best performance, it reduces 54% in average the total simulation time with respect to $\texttt{PCP}_4$ on KIT. Further, $\texttt{PCP}_5$ presents the minimum average dispatching time using all predictors, however, both dispatchers generate dispatching decisions in less than 1 sec on average, which is valuable given the size of the system. Although, $\texttt{PCP}_5$ presents the best timings, we realized that during the simulation at different complex instances, that is when the system usage and the number of running and queued jobs are high, $\texttt{PCP}_5$ employs up to 20x memory RAM than $\texttt{PCP}_4$, during the simulation of KIT . Therefore, $\texttt{PCP}_5$ is limited by the capacity of the actual system on which runs or the capacity of the CP solver to manage the underlying data structures to deal with complex instances.

Looking at the quality of the dispatching decision, in terms of slowdown and waiting time, in Figure 6.6, their trends are similar as the results presented in the Chapter 5. $\texttt{PCP}_4$ and $\texttt{PCP}_5$ improve the quality of their dispatching decision of good predictors, in this case of `-R` with respect to `-W`. `-L2` seems to put the quality of decisions at risk, since their results are worst than `-W`. We recall we use this prediction method because `-D` cannot be used given the available data from the workload dataset. The results obtained by $\texttt{PCP}_4$ and $\texttt{PCP}_5$ on the KIT system are similar, however, the dispersion of the first dispatcher is lower using `-W` and `-R`, whereas using `-L2` seems to be equal.

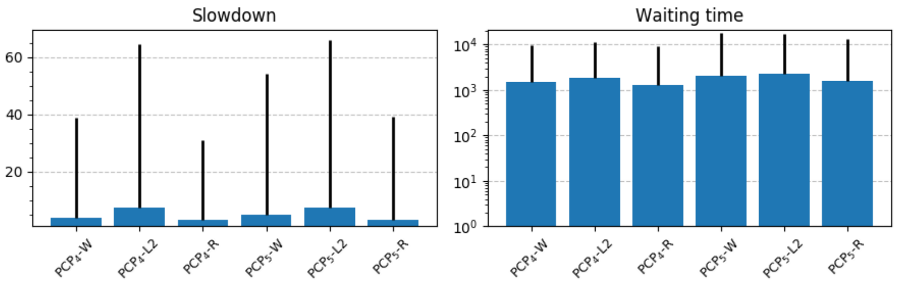


Figure 6.6: Average slowdown and waiting time of $\texttt{PCP}_4$ and $\texttt{PCP}_5$ on the KIT system.

6.5.2 $\texttt{PCP}_4$ and $\texttt{PCP}_5$ on the Eurora system

We already know that the proposed dispatchers can manage the job requests on big systems, however, we need to evaluate how these dispatchers behave on small systems, for which $\texttt{PCP}_3$ presents outstanding results. One important aspect is the number of decision variables involved in the model, for which, we claimed that $\texttt{PCP}_4$ and $\texttt{PCP}_5$ reduce them with respect to those used in by $\texttt{PCP}_3$. Using the same instances across $\texttt{PCP}_3$, $\texttt{PCP}_4$ and $\texttt{PCP}_5$, we show in Figure 6.7, the ratio of the number of decision variables used in a proposed dispatcher regarding $\texttt{PCP}_3$. In total there are instances 624,564, which corresponds to the simulation of the Eurora workload dataset using $\texttt{PCP}_3$ with `-D` and `-R` prediction methods. We placed together the results of both simulations because we are interested in a value which is independent of the dispatching quality. This figure shows a reduction in the number of decision variables of 2% of the total instances of $\texttt{PCP}_4$ with respect to $\texttt{PCP}_3$. Instead, $\texttt{PCP}_5$ reduces the number of variables considerably for almost 99% of the instances. So, the reduction of the number of decision variables may explain the reduction in the total dispatching time presented in the previous experiments, for which $\texttt{PCP}_5$ showed the best results in terms of a fast response.

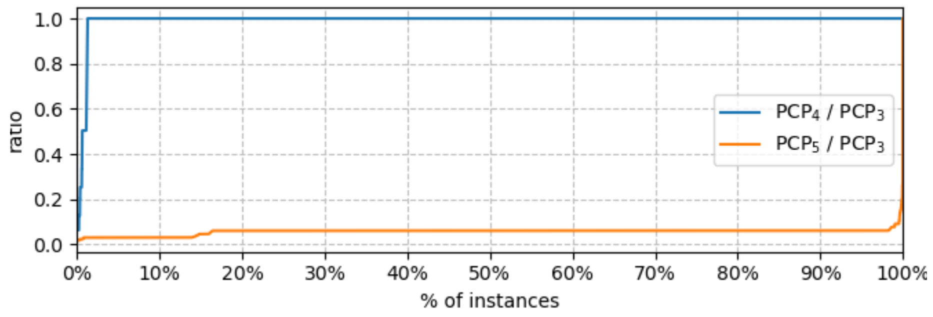


Figure 6.7: Ratio of the number of decision variables.

Incurred time to generate dispatching decisions We evaluate the required time for generating a dispatching decision, which corresponds to the incurred time of a dispatcher for modeling and solving an instance. The incurred time to generate a dispatching decision is a fundamental feature of HPC job dispatchers, for the same reason that heuristics are used, thus a job dispatcher must ensure a quick response with a an high quality decision.

Figure 6.8 shows the ratio between the incurred time for generate a dispatching decision for an instance between $\mathtt{PCP}_4$ and $\mathtt{PCP}_3$, and $\mathtt{PCP}_5$ and $\mathtt{PCP}_3$. We use the ratio of the incurred time between a proposed dispatcher and the $\mathtt{PCP}_3$, so a ratio less than 1 means the required time for generating a dispatching decision was reduces, otherwise it is increased. The figure shows a high improvement by the $\mathtt{PCP}_5$ dispatcher, which reduces the incurred time to generate a dispatching decision regarding $\mathtt{PCP}_3$for more than 95% of the instances. The results obtained by $\mathtt{PCP}_4$ are not conclusive because for 50% of the instances it reduces the incurred time and for the other half it increases using `-R` instead `-D` presents a reduction for 70% of the instances.

We believe that using the real duration presents a more complex scenario because the variability of job durations is higher, which presents more options to elaborate a dispatching decision, whereas `-D` in some cases use similar durations reducing the number of possible solutions because the minimization of the objective function depends on this job feature.

Quality of dispatching decisions In Figures 6.9 and 6.10, we compare the quality of dispatching decisions in terms of the value objective function of the best solution found for each instance solved by $\mathtt{PCP}_4$ and $\mathtt{PCP}_5$ both regarding $\mathtt{PCP}_3$, respectively. In the figures, each point corresponds to the value of the objective function of an instance solved by one of the new proposed dispatchers (x value) and $\mathtt{PCP}_3$ (y value). In addition, each color has a specific meaning: A green point corresponds to an instance on which the proposed dispatcher beats $\mathtt{PCP}_3$. A blue point represents an identical objective function value of both solutions. Red points correspond to instances on which $\mathtt{PCP}_3$ is better. We recall the objective function of the dispatchers is the minimization of the

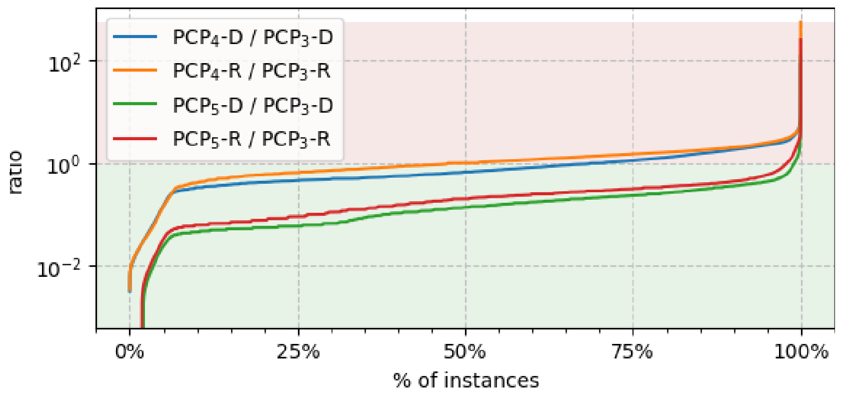
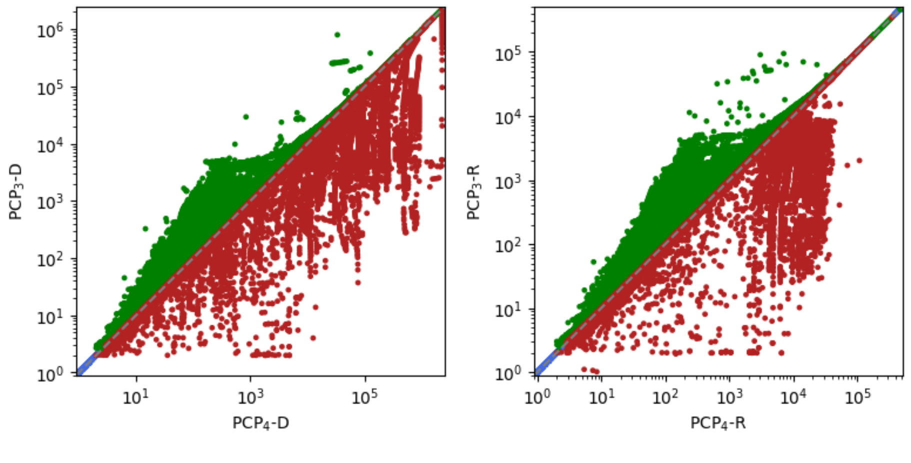


Figure 6.8: Ratio of the required time to generate a dispatching decision on the Eurora system.

jobs' slowdown, so the minimum value is $1.0 * |\hat{Q}|$. Although the results seem to benefit $\texttt{PCP}_3$, the percentage of the colored points present in Table 6.2 shows a different scenario. Green points, i.e. better O.F., correspond to twofold the red ones, that is worse O.F., for both comparisons and using `-D` and `-R`.

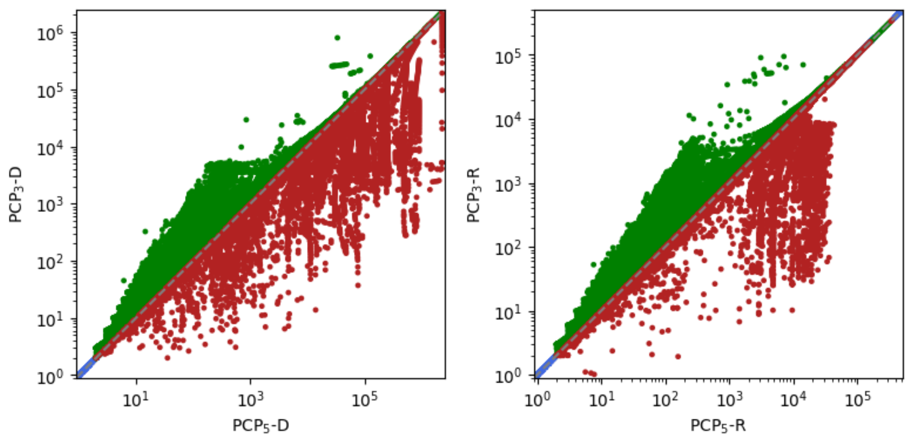
For a given instance, if there is no solution given the time limit, the dispatcher does not provide a dispatching decision and continues with the next one, therefore the instance is considered as not solved. In general, the instances not solved by $\texttt{PCP}_3$ are highly reduced by $\texttt{PCP}_4$ and increased by $\texttt{PCP}_5$.



(a) $\texttt{PCP}_4\texttt{-D}$ vs $\texttt{PCP}_3\texttt{-D}$ (b) $\texttt{PCP}_4\texttt{-R}$ vs $\texttt{PCP}_3\texttt{-R}$

Figure 6.9: Comparison of the quality of the solution of instances of $\texttt{PCP}_4$ w.r.t $\texttt{PCP}_3$.

We compare the quality of the solution for each instance using the ratio of



(a) `PCP`$_5$`-D` vs `PCP`$_3$`-D` (b) `PCP`$_5$`-R` vs `PCP`$_3$`-R`

Figure 6.10: Comparison of the quality of the solution of instances of `PCP`$_5$ w.r.t `PCP`$_3$.

	`PCP`$_4$`-D`	`PCP`$_3$`-D`	`PCP`$_4$`-R`	`PCP`$_3$`-R`	`PCP`$_5$`-D`	`PCP`$_3$`-D`	`PCP`$_5$`-R`	`PCP`$_3$`-R`
Total instances	311,506		312,488		311,178		311,763	
Better O.F.	23,427 (8%)		21,498 (7%)		23,734 (8%)		20,403 (7%)	
Equal O.F.	275,457 (88%)		279,513 (89%)		275,158 (88%)		279,662 (89%)	
Worse O.F.	12,622 (4%)		11,477(4%)		12,286 (4%)		11,698 (4%)	
Not solved	6	254	38	302	381	254	823	302

Table 6.2: Comparison of the quality of the decisions of the instances solved by `PCP`$_4$`-R` and `PCP`$_5$`-R` with respect to `PCP`$_3$`-R` on the Eurora system.

the value of the objective function of the best solution found between a proposed dispatcher and `PCP`$_3$. Given that the dispatchers are intended to minimize the objective function, a ratio below 1 corresponds to an improvement in the quality of the solution for a given instance, whereas a ratio above 1 corresponds to the opposite case. Using the ratio of quality, we can easily appraise the improvement or diminishment of the quality of the solutions for each individual instance. Figure 6.11 combines the ratios using both dispatchers with the two prediction methods. The figure on the left shows the instances for which `PCP`$_4$ and `PCP`$_5$ improved the quality of solutions generated by `PCP`$_3$. The improvement of `PCP`$_4$`-R` and `PCP`$_5$`-R` is at least 50% for 2% of the total instances, instead `PCP`$_4$`-D` and `PCP`$_5$`-D` dispatchers reached the same improvement for 1.5% of the total instances. Since the objective function is based on the job duration, the proposed dispatchers show the importance of the accuracy of the prediction of job durations, as we described in Chapter 5. Conversely, worsening the solution 50% or more only occurs on 1% for the `PCP`$_4$`-R` and `PCP`$_5$`-R`, instead 2% for `PCP`$_4$`-D` and `PCP`$_5$ regarding `PCP`$_3$`-D`. The `PCP`$_5$`-R` / `PCP`$_3$`-R` shows a marginal improvement in the results with respect to the `PCP`$_4$`-R` / `PCP`$_3$`-R` ratio by reducing the gap in the results of the figure on the right.

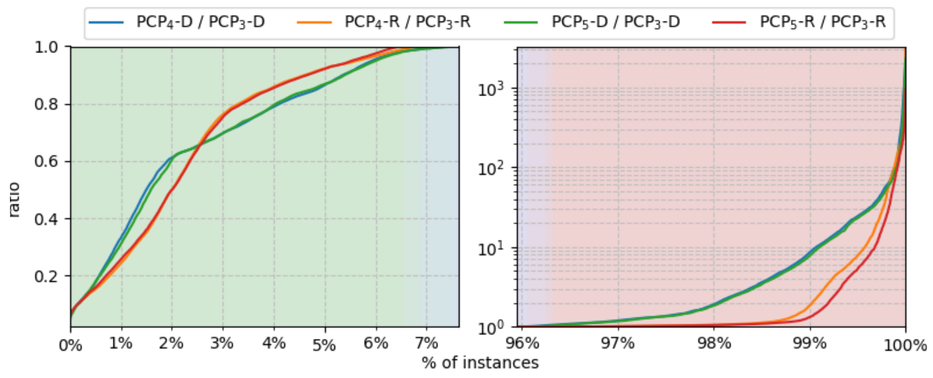


Figure 6.11: Ratio of the value of the O.F. between a proposed dispatcher and $\texttt{PCP}_3$ with `-D` and `-R` of all instances on the Eurora system.

Characterization of instances We analyzed previously the quality and the incurred time to generate dispatching decisions, however, both evaluations were analyzed separately. We already know, that on average, 88% of the instances present the same quality and at least 50% of the instances reduced the incurred time to generate a dispatching decision, but which are those instances?

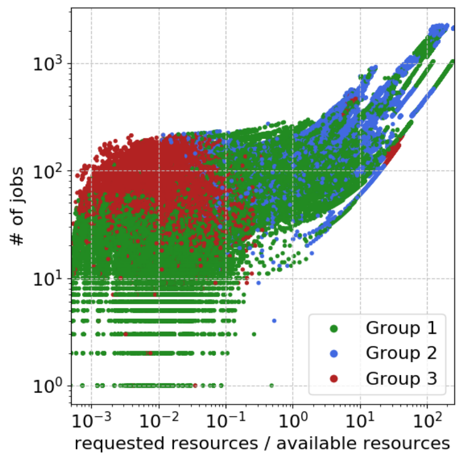
We characterized the instances in order to determine when the proposed dispatchers are a good alternative to $\texttt{PCP}_3$, considering two important aspects of an instance such as number of jobs (y-axis) and the ratio between the requested and the available resources (x-axis); and, in addition, the combination of the following indicators using a color: Better OF, Same OF, and Worse OF; and Better time, Same time and Worse time. We mean for OF as the value of the objective function for the best solution found and time as the required time generating the dispatching decision, both for a given instance. Therefore, an indicator is the result of the comparison of a proposed dispatcher with $\texttt{PCP}_3$ in terms of OF and time. Some indicators are grouped as defined in Table 6.3. For instance, if the result of a given instance is Better OF and Better time, given both indicators, this result corresponds to Group 1. Each group has a specific meaning and represents in which scenario a proposed dispatcher or $\texttt{PCP}_3$ should be selected as the system dispatcher. Group 1 includes all possible combinations on which the proposed dispatcher represents to be the better alternative as the job dispatcher of the system. Instead, group 2 consists of all combinations for which one indicator should be sacrificed, either the quality of the solution or time to generate a dispatching decision. Finally, group 3 includes all combinations for which the proposed dispatcher should not be used.

Figure 6.12 shows the results of $\texttt{PCP}_4$`-D` regarding $\texttt{PCP}_3$`-D` in the figure on the left, whereas in the right one, the results obtained by $\texttt{PCP}_4$`-R` regarding $\texttt{PCP}_3$`-R`. In general, $\texttt{PCP}_4$ presents more instances in Group 1, which means that $\texttt{PCP}_4$ outperforms in quality and time $\texttt{PCP}_3$. Although $\texttt{PCP}_4$`-R` shows lower performance

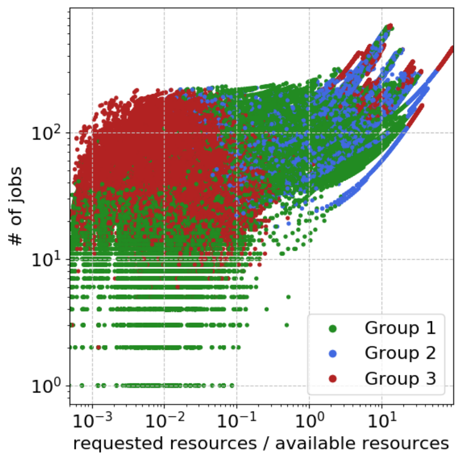
	Indicators	$\frac{\texttt{PCP}_4\texttt{-D}}{\texttt{PCP}_3\texttt{-D}}$	$\frac{\texttt{PCP}_4\texttt{-R}}{\texttt{PCP}_3\texttt{-R}}$	$\frac{\texttt{PCP}_5\texttt{-D}}{\texttt{PCP}_3\texttt{-D}}$	$\frac{\texttt{PCP}_5\texttt{-R}}{\texttt{PCP}_3\texttt{-R}}$
Group 1	Better OF and time, Better OF and same time, same OF and better time, and same OF and time	67.1%	49.6%	95.5%	95.0%
Group 2	Better OF and worse time, and worse OF and better time	3.9%	3.8%	3.9%	3.8%
Group 3	Worse OF and time, same OF and worse time, worse OF and equal time	29.0%	46.6%	0.6%	1.2%

Table 6.3: Percentage of instances from the Eurora system grouped by performance indicators

than PCP$_4$-D, the results still are better than PCP$_3$-R. The lower performance is explained by the required time for generating a dispatching decision which is slightly higher than the one required by PCP$_4$-D(we showed previously in Figure 6.8). The main disadvantage of PCP$_4$ with respect PCP$_3$ appears on instances in which the number of requested resources is less than the available, i.e. when the system is not fully loaded, and when the number of queued jobs is higher than 30, which is represented in the area where more red points (Group 3) are concentrated on the figures. Conversely, if the number of queued jobs is reduced, or the requested resources go towards the availability of the resources PCP$_4$ shows its better performance (Group 1). Group 2 instances seem to be very complex instances, many jobs, and more requested resources than the available ones, however these instances represent a small percentage of the total.



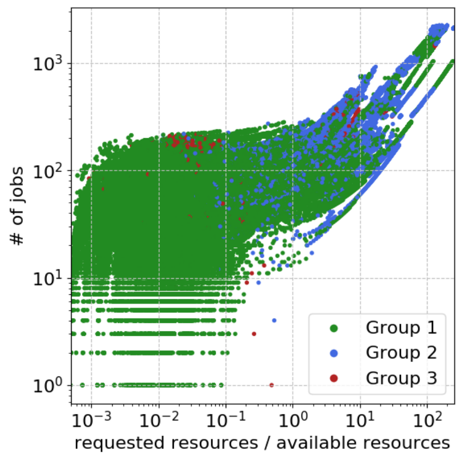
(a) PCP$_4$-D vs PCP$_3$-D



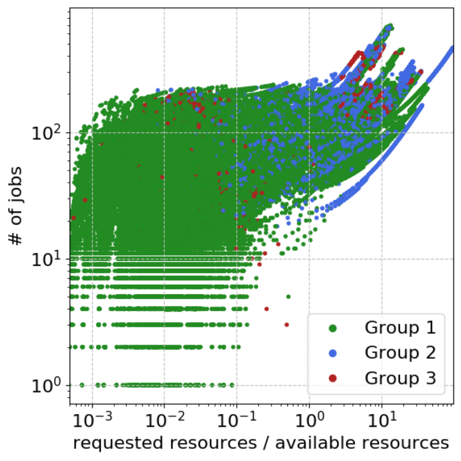
(b) PCP$_4$-R vs PCP$_3$-R

Figure 6.12: Characterization of the performance of PCP$_4$ regarding PCP$_3$ on all instances

Figure 6.13 shows the results of PCP$_5$ regarding PCP$_3$. The results are categorical, PCP$_5$ presents the best results regarding quality of dispatching and time to generate a dispatching overall all instances with respect to PCP$_3$. Despite there are some results of instances that fall into Group 3, these instances represent in average 0.9% of the total. Most of the results of the instances corresponds to Group 1, which represent in general, a very fast response with a high quality dispatching decision. Finally, the percentage of results of the instances corresponding to Group 2 are similar to those achieved by $\texttt{PCP}_4$. Although $\texttt{PCP}_5$ seems to be the best dispatcher to work with Eurora, in average 4.5% of the instances corresponds to a 'sacrifice area', where a dispatcher can be good to achieve a good quality with a slow response or the contrary, thus, we believe that a future HPC dispatcher should be capable to decide which model to use based on different attributes of the instance to tackle it. The instances in Group 3, around 0.9% in average, are characterized by the symmetry of the request and availability of resources, increasing the size of the search tree by revisiting equivalent states. Proper symmetry breakers may be used to potentially reduce the search space and improve the performance of the dispatchers, and consequently, find the same solutions in less time, nicer still find better solutions in the same time limit.



(a) $\texttt{PCP}_5\texttt{-D}$ vs $\texttt{PCP}_3\texttt{-D}$



(b) $\texttt{PCP}_5\texttt{-R}$ vs $\texttt{PCP}_3\texttt{-R}$

Figure 6.13: Characterization of the performance of $\texttt{PCP}_5$ regarding $\texttt{PCP}_3$ on all instances

6.6 Summary

We proposed two new CP-based dispatchers that use a *single CP model* for the on-line job dispatching problem in HPC systems. Such models may allow the integration of more restrictions on real-world HPC systems, which will be needed to cope with the current needs of jobs and resources management. A *single CP model* also allows the generation of solid dispatching decisions and also the integration of better allocation strategies. We studied the proposed dispatchers and compared them with the best dispatcher from Chapter 5 of this *single CP model* class. However, we showed that $\texttt{PCP}_3$ could not handle jobs in a big system, for which $\texttt{PCP}_4$ and $\texttt{PCP}_5$ could. Therefore, we evaluated the proposed dispatchers in a small system such as Eurora, where $\texttt{PCP}_3$ showed good performance. The results were also categorical, especially with the $\texttt{PCP}_5$ dispatcher, beating in terms of QoS and reducing the time to generate a dispatching decision with respect to the results obtained the $\texttt{PCP}_3$ dispatcher. It is worthwhile to mention that we significantly reduced incurred time for generating decisions with both proposed dispatchers. In addition, $\texttt{PCP}_4$ and $\texttt{PCP}_5$ can be used as a dispatcher on big systems, and do not suffer from scalability issues in such systems.

Although $\texttt{PCP}_5$ may be a valid alternative to deal with the on-line dispatching problem on almost any-size systems, there are still instances for which it does not present the best results and the higher number of non-solved instances. Therefore, $\texttt{PCP}_5$ should be improved at least to reduce the high number of non-solved instances by improving the search strategy or including the proper symmetry breakers. Even though $\texttt{PCP}_5$ is improved, we believe a dispatcher should be composed of many *single CP models*, and during dispatching time a model should be selected based on the instance features. This would lead to cover many more instances during runtime, for which the performance of a *single CP model* is unknown and can be replaced by alternative models that can show better results during runtime. Thus, as future work, we plan to develop a “hyper-dispatcher”, which based on the features of an instance, selects a specific model to use, for which it has a track of success on such class of instance.

Chapter 7

Related work

This chapter gives an overview on the existing research related to two concepts treated in Chapters 4, 5, and 6. Section 7.1 explores different approaches to predict different attributes of jobs on HPC systems. Instead, Section 7.2 presents different approaches used to solve the (on-line) job dispatching problem in HPC systems.

7.1 Prediction on HPC workloads

A number of previous efforts have developed techniques for predicting interesting aspects of workloads such as power consumption and job duration [109, 123]. Borghesi et al. [16] propose a machine learning approach to forecast the mean power consumption of HPC applications using only information available at scheduling time, such as the resources requested, the maximum duration, the user, etc. Sirbu et al. [112] present a support vector machine model to predict the power consumption of jobs, taking also into account their variability.

Predicting the durations of HPC jobs have also been considered in previous research works, especially in relation to job dispatching [88, 31]. Tsafrir et al. [129] propose a model that uses the run times of the last two jobs to predict the duration of the next job. This prediction is then used for scheduling purposes. Their approach is lightweight and efficient, however, the prediction accuracy can be improved using more complex techniques like the ones proposed in this paper. Gaussier et al. [52] show the importance of estimating the duration of HPC jobs with backfilling schedulers. Their results clearly suggest that a backfilling policy benefits from accurate duration predictions, indeed, a simple linear model can improve the slowdown of backfilling techniques by 28%.; the only limitation is that their work focuses exclusively on a particular scheduling algorithm.

Recently, machine learning methods were applied to predict job duration [67, 118], including metadata such as job names as features. However, it require model training and the prediction were not integrated within a dispatcher for testing to understand the overhead during execution.

Underestimation of job duration is a problem that appears often in the literature, since it negatively affects dispatcher performance, more than overestimation. Recently, [42] proposed a predictive method based on a censored regression model, which could minimize underestimation. Although promising, it requires heavier computations.

The heuristic methods for job duration prediction proposed in this work relied on matching the metadata of a current job with similar metadata of finished jobs. Similar to [129], which bases the prediction of previous durations, we associate different matching rules and use a previous duration. The heuristic methods proposed here are simpler, and they do not require model training.

7.2 HPC job dispatchers

A job dispatcher is composed of two main tasks, the job scheduler and the job allocator, which selects queued jobs to be allocated and allocate it to the system resources. Queue-based scheduling is the most common way in which jobs are scheduled. Commercial systems such as SLURM or PBS use this approach together with different queues for which a priority, restrictions of the requests, or other properties are assigned. The objective of a queue-based scheduler is to select waiting jobs following the queue order. A well-known strategy is First Come, First Served (or First-In, First-Out), FCFS, which selects jobs according to their submission times [2]. Other strategies assign priorities based on certain job attributes, such as, the expected duration, e.g. Short Job First (SJF) or Longer Job First (LJF), however, these strategies rely on the job's walltime [2]. However, these job schedulers suffer from starvation when a job in the queue cannot be dispatched (blocked job), the other queued job will remain in the queue until the blocked job is dispatched, which may cause a low system utilization [44].

An improved version of simple schedulers such as FCFS, SJF or LJF, is the Backfilling scheduler [82]. Backfilling tries to increase the utilization while maintaining the queue order prioritization (FCFS, SFJ, etc.). In general, it allows small jobs to move ahead and utilize the system that would otherwise remain idle. Backfilling has been considered as the best queue-based scheduling algorithm, and many efforts has been put on it [126, 134, 68, 33, 120]. Despite all this effort, all backfilling algorithms consider the queued jobs one at a time, and together with an allocation algorithm, try to dispatch them, which may cause fragmentation of the system. To tackle this issue, it is necessary to consider the entire queue at once, and generate a dispatching decision that satisfies the system stakeholders' expectations. This can be done using plan-based dispatching. Such dispatchers generate a dispatching decision based on a plan considering the present and future of the system, so all queued jobs will be scheduled between the current time and some time horizon. Every submitted job is planned immediately, and if a running job ends before it was estimated to end, a new dispatching decision is generated. Plan-based dispatchers are based on optimization methods, so the job management is treated as a optimization problem, where it is necessary to optimize the HPC system, minimizing or maximizing one of its many objectives or performance criteria. There are dispatchers based on metaheuristics, such as Global Optimising Resource Broker and Allocator (GORBA) [125] which uses an Evolutionary Algorithm to solve the on-line job dispatching problem in Grid Environments. Klusacek et At. [72] proposes a combination an heuristic to build a starting solution which later is tried to be improved using the Tabu search metaheuristic. In general, metaheuristics have been shown that can provide efficient dispatching results than traditional dispatching algorithms [71, 111]. State-of-the-art dispatchers based on Constraint Programming (CP) has been also shown outstanding results regarding traditional dispatching algorithms [25, 18]. CP-based dispatchers are present in two different approaches (i) a pure CP model and (ii) a hybrid CP model, where the first approach models with a single model the scheduling and allocation problem, the second one models only the scheduling problem and integrates it with a well-known resource allocation strategy, named Best-Fit, using prioritization rules, which later was improved using other allocation strategies [93]. Hybrid models were proposed to improve the scalability to heavy workloads because pure CP models tend to require higher CPU-times concerning queue-based scheduling algorithms; however, the synergy of a single model is lost.

None of these works combine job duration prediction with a CP-based job dispatcher as the presented in this research work. We did it initially with the hybrid CP model, `HCP`, however, it was done naively by replacing the expected durations with predicted durations, and the results were not satisfactory, leading to worse performance compared to the wall-time approach. Analyzing the results, we realized an issue related to the underestimation. Later, we applied again to `HCP`, and to a pure CP model, `PCP`, adapting the model and search of both dispatchers to deal with duration underestimation and to the use of predictions. The results showed an improvement of the QoS in terms of slowdown and waiting time of `HCP` regarding the use of the wall-time. Instead, the simulations using `PCP` showed some issues regarding the scalability to heavy workloads, where this dispatcher could not process the queue during a limited time to generate a dispatching decision. So new dispatchers $\texttt{HCP}_3$ and $\texttt{PCP}_3$, based on top `HCP` and `PCP` respectively, were proposed to make them suitable to heavy workloads and showed improvements regarding their former versions.

Despite this, $\texttt{PCP}_3$ still presented issues on big systems (systems with hundreds of nodes). We identified the main issue as to how $\texttt{PCP}_3$ models the decision variables because they increase in number with respect to the number of nodes of the system. Thus, we proposed $\texttt{PCP}_4$ and CPyvar which are less dependant on the number of nodes of a system. Thus these new dispatchers are more suitable for big systems. $\texttt{PCP}_4$ and $\texttt{PCP}_5$ handled the job submission on big systems, and in addition, these dispatchers presented competitive results also on smaller systems, where $\texttt{PCP}_3$ presented outstanding results.

Chapter 8

Conclusions and future work

HPC systems are closer to reach the exascale computation (10^{18} operations per sec.), while we can expect progress in hardware design to be a major contributor towards these goals, rest of the increase has to come from software techniques and from massive parallelism employing millions of processor cores, hence a Workload Management System is a fundamental software available on any HPC system. Major improvements in the system performance are generated by the definition of better components, especially in the job dispatcher. Job dispatching strategies become critical for keeping system utilization high while keeping waiting times low, even better if the slowdown is also low, for jobs that are competing for HPC system resources.

As we have shown, CP-based dispatchers proved to be an outstanding alternative to classic queued-based dispatchers. However, doing research in this field is not trivial because a lot of experimentation should be carried to cover all aspects of HPC systems that occur on-line. We mean for online in the sense that decisions are made during real-time and there is no knowledge about the future, and a current decision alters the state of later decisions. To control the experiments and be able to repeat, we developed a WMS simulator, which was used in the context of our research. We developed Accasim, a simulator that scales well to large workload datasets and is easy to customize, so allowing to carry out experiments across different workload sources, resource types, and dispatching algorithms.

Ideally, dispatching decisions should complete all jobs in the shortest amount of time possible, and maintaining high Quality of Service (QoS) levels while keeping the system utilization high. The job slowdown is a very important QoS metric used in HPC systems. We recall this metric is the normalization of the waiting time of a job regarding its job duration, and the waiting time is the time that passes between a job is submitted to the system and when it starts to run. Therefore, this metric is highly valuable by HPC users, that do not want to wait long if the jobs are short. This metric turns more interesting, after analyzing all the available workloads datasets that we had access to. All of these workloads are mainly composed of short jobs. However, maintaining high QoS levels is achievable only with complete a priori knowledge of the workload, which is impossible with the nature of the on-line dispatching problem, and the actual duration of a job seems to be an important asset. Indeed, most of the dispatchers available on literature assume that job durations are known in advance, whereas, in reality, the requested wall-time, which is only an upper bound for this value is known. For those dispatchers, the requested wall-time is often used as the expected duration. Thus, in order to improve the QoS of a system that uses a dispatcher that bases its goals on job duration metrics, we had to improve the expected durations based on wall-time because users tend to overestimate it, and an overestimation may cause inconsistent decisions. An accurate prediction may improve substantially dispatching decisions, thus improving system performance. Therefore, we proposed two data-driven approaches which showed to be more effective predictions than the estimates based on wall-time. The use of our data-driven job duration predictors showed higher QoS levels in terms of job waiting times and slowdown regarding the use of the wall-time.

Even if the durations are known, current dispatchers do not take advantage of it nor explore all possible solutions searching for the optimum. CP-based dispatchers have been proposed to address these issues, however, they are unable to satisfy the challenges of on-line dispatching. Therefore, we built on top state-of-the-art CP-based dispatchers and redesign their main components. We revisited their model and search control mechanism so as to make them resilient to heavy workloads and applicable to on-line dispatching. Moreover, we integrated and adapted the model and search algorithm of our dispatchers to the use of job duration predictions to obtain high QoS levels in terms of job waiting times and slowdown. This improvement is achieved especially for workloads dominated by short jobs, and only if expected durations rely on predictions with acceptable levels of accuracy, going beyond the standard wall-time approach. The quality of the decisions generated by the proposed CP-based dispatcher, $\texttt{PCP}_3$ and $\texttt{HCP}_3$, are prominent and also most of them showed a reduction of the average time incurred to generate a dispatching decision. We also showed that giving priority to short jobs does not penalize, in general, medium and long jobs.

However, $\texttt{PCP}_3$ and $\texttt{HCP}_3$ have been analyzed with respect a small and medium system. Therefore, we asked ourselves the following question: How these dispatchers will behave on a bigger system? Analyzing the models, we know that $\texttt{HCP}_3$ may scale well on big systems because it only considers one variable for each queued job in $\bar{Q}$. Despite this, its hybrid structure may hideout all the CP potential by generating weak dispatching decisions based on an uncoupled approach. Instead, CP-based dispatchers must be structured into a *single CP model* that consider the scheduling and allocation problems together, which may be easier to incorporate additional restrictions and provide better dispatching decisions for current and future HPC systems. However, the *single CP model* of $\texttt{PCP}_3$, presented a scalability issue in big systems. Thus, we proposed two new CP-based dispatchers, $\texttt{PCP}_4$ and $\texttt{PCP}_5$, that use a *single CP model* to model the on-line job dispatching problem. These new dispatchers reduce the number of decision variables with respect to $\texttt{PCP}_3$, thus are more suitable to model instances of big HPC systems. We obtained outstanding results regarding $\texttt{PCP}_3$, especially $\texttt{PCP}_5$which beat in terms of QoS and reducing the time to generate a dispatching decision on a small system and moreover can handle job requests in a big system. Although $\texttt{PCP}_5$ may be a valid alternative to deal with the on-line dispatching problem on almost any-size systems, there are still instances for which it does not present the best results and the higher number of non-solved instances regarding $\texttt{PCP}_3$. Instead, $\texttt{PCP}_4$ shows less number of unsolved instances than $\texttt{PCP}_5$ but overall showed a similar performance that $\texttt{PCP}_3$ on the solved ones.

The contribution of this research is a step forward to the adoption of CP-based dispatchers in actual HPC systems thanks to reducing the timings and improving the QoS in terms of slowdown and waiting time. We believe that selecting a CP model for a system is not a answer for actual HPC systems, instead, a CP-based dispatcher should be composed of many CP models, and depending on the features of the instances, a CP model should be selected. This is because users on HPC systems change during time, as well as their submissions, also systems are modified or are affected by disruptions of their components. A dispatcher for future HPC systems should be able to auto-reconfigure their parameters and strategies given the instances and learn from past decisions. Thus, as future work, we plan to develop a “hyper-dispatcher”, which based on the features of an instance, selects a specific model to use, for which it has a track of success on such class of instance. Current and future HPC systems require to optimize more objectives and support more constraints than currently considered in this work, such as power management and computational overhead. We believe such constraints can be completely adopted in a *single CP models*, such as the proposed here, to lead to the efficient management of jobs and resources.

Bibliography

[1] Bilge Acun, Nikhil Jain, Abhinav Bhatele, Misbah Mubarak, Christopher D. Carothers, and Laxmikant V. Kalé. Preliminary evaluation of a parallel trace replay tool for HPC network simulations. In Sascha Hunold, Alexandru Costan, Domingo Giménez, Alexandru Iosup, Laura Ricci, María Engracia Gómez Requena, Vittorio Scarano, Ana Lucia Varbanescu, Stephen L. Scott, Stefan Lankes, Josef Weidendorfer, and Michael Alexander, editors, *Euro-Par 2015: Parallel Processing Workshops - Euro-Par 2015 International Workshops, Vienna, Austria, August 24-25, 2015, Revised Selected Papers*, volume 9523 of *Lecture Notes in Computer Science*, pages 417–429. Springer, 2015.

[2] Kento Aida. Effect of job size characteristics on job scheduling performance. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing, IPDPS 2000 Workshop, JSSPP 2000, Cancun, Mexico, May 1, 2000, Proceedings*, volume 1911 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 2000.

[3] Altair. Altair PBS professional, 2019.

[4] Axel Auweter, Arndt Bode, Matthias Brehm, Luigi Brochard, Nicolay Hammer, Herbert Huber, Raj Panda, Francois Thomas, and Torsten Wilde. A case study of energy aware scheduling on supermuc. In Julian Martin Kunkel, Thomas Ludwig, and Hans Werner Meuer, editors, *Supercomputing - 29th International Conference, ISC 2014, Leipzig, Germany, June 22-26, 2014. Proceedings*, volume 8488 of *Lecture Notes in Computer Science*, pages 394–409. Springer, 2014.

[5] Ayan Banerjee, Tridib Mukherjee, Georgios Varsamopoulos, and Sandeep K. S. Gupta. Integrating cooling awareness with thermal aware workload placement for HPC data centers. *SUSCOM*, 1(2):134–150, 2011.

[6] Philippe Baptiste, Philippe Laborie, Claude Le Pape, and Wim Nuijten. Constraint-based scheduling and planning. In Francesca Rossi, Peter van Beek, and Toby Walsh, editors, *Handbook of Constraint Programming*, volume 2 of *Foundations of Artificial Intelligence*, pages 761–799. Elsevier, 2006.

[7] Philippe Baptiste, Claude Le Pape, and Laurent Péridy. Global constraints for partial csps: A case-study of resource and due date constraints. In Michael J. Maher and Jean-Francois Puget, editors, *Principles and Practice of Constraint Programming - CP98, 4th International Conference, Pisa, Italy, October 26-30, 1998, Proceedings*, volume 1520 of *Lecture Notes in Computer Science*, pages 87–101. Springer, 1998.

[8] Roman Barták. Principles of constraint processing. In *Artificial Intelligence for Advanced Problem Solving Techniques*, pages 63–106. IGI Global, 2008.

[9] Andrea Bartolini, Andrea Borghesi, Thomas Bridi, Michele Lombardi, and Michela Milano. Proactive workload dispatching on the EURORA supercomputer. In Barry O'Sullivan, editor, *Principles and Practice of Constraint Programming - 20th International Conference, CP 2014, Lyon, France, September 8-12, 2014. Proceedings*, volume 8656 of *Lecture Notes in Computer Science*, pages 765–780. Springer, 2014.

[10] Andrea Bartolini, Matteo Cacciari, Carlo Cavazzoni, Giampietro Tecchiolli, and Luca Benini. Unveiling eurora - thermal and power characterization of the most energy-efficient supercomputer in the world. In Gerhard P. Fettweis and Wolfgang Nebel, editors, *Design, Automation & Test in Europe Conference & Exhibition, DATE 2014, Dresden, Germany, March 24-28, 2014*, pages 1–6. European Design and Automation Association, 2014.

[11] Nicolas Beldiceanu and Evelyne Contejean. Introducing global constraints in CHIP. *Mathematical and Computer Modelling*, 20(12):97 – 123, 1994.

[12] Christian Bessiere. Constraint propagation. In Francesca Rossi, Peter van Beek, and Toby Walsh, editors, *Handbook of Constraint Programming*, volume 2 of *Foundations of Artificial Intelligence*, pages 29–83. Elsevier, 2006.

[13] Christian Bessière and Jean-Charles Régin. MAC and combined heuristics: Two reasons to forsake FC (and cbj?) on hard problems. In Eugene C. Freuder, editor, *Proceedings of the Second International Conference on Principles and Practice of Constraint Programming, Cambridge, Massachusetts, USA, August 19-22, 1996*, volume 1118 of *Lecture Notes in Computer Science*, pages 61–75. Springer, 1996.

[14] Jacek Blazewicz, Jan Karel Lenstra, and A. H. G. Rinnooy Kan. Scheduling subject to resource constraints: classification and complexity. *Discrete Applied Mathematics*, 5(1):11–24, 1983.

[15] Deva Bodas, Justin Song, Murali Rajappa, and Andy Hoffman. Simple power-aware scheduler to limit power consumption by HPC system within a budget. In Kirk W. Cameron, Adolfy Hoisie, Darren J. Kerbyson, David K. Lowenthal, Dimitrios S. Nikolopoulos, Sudha Yalamanchili, and

Andres Marquez, editors, *Proceedings of the 2nd International Workshop on Energy Efficient Supercomputing, E2SC '14, New Orleans, Louisiana, USA, November 16-21, 2014*, pages 21–30. IEEE Computer Society, 2014.

[16] Andrea Borghesi, Andrea Bartolini, Michele Lombardi, Michela Milano, and Luca Benini. Predictive modeling for job power consumption in HPC systems. In Julian M. Kunkel, Pavan Balaji, and Jack J. Dongarra, editors, *High Performance Computing - 31st International Conference, ISC High Performance 2016, Frankfurt, Germany, June 19-23, 2016, Proceedings*, volume 9697 of *Lecture Notes in Computer Science*, pages 181–199. Springer, 2016.

[17] Andrea Borghesi, Andrea Bartolini, Michele Lombardi, Michela Milano, and Luca Benini. Scheduling-based power capping in high performance computing systems. *SUSCOM*, 19:1–13, 2018.

[18] Andrea Borghesi, Francesca Collina, Michele Lombardi, Michela Milano, and Luca Benini. Power capping in high performance computing systems. In Gilles Pesant, editor, *Principles and Practice of Constraint Programming - 21st International Conference, CP 2015, Cork, Ireland, August 31 - September 4, 2015, Proceedings*, volume 9255 of *Lecture Notes in Computer Science*, pages 524–540. Springer, 2015.

[19] Frédéric Boussemart, Fred Hemery, Christophe Lecoutre, and Lakhdar Sais. Boosting systematic search by weighting constraints. In Ramón López de Mántaras and Lorenza Saitta, editors, *Proceedings of the 16th Eureopean Conference on Artificial Intelligence, ECAI'2004, including Prestigious Applicants of Intelligent Systems, PAIS 2004, Valencia, Spain, August 22-27, 2004*, pages 146–150. IOS Press, 2004.

[20] Jim M. Brandt, Bert J. Debusschere, Ann C. Gentile, Jackson Mayo, Philippe P. Pébay, David C. Thompson, and Matthew Wong. Using probabilistic characterization to reduce runtime faults in HPC systems. In *8th IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2008), 19-22 May 2008, Lyon, France*, pages 759–764. IEEE Computer Society, 2008.

[21] Tracy D. Braun, Howard Jay Siegel, Noah Beck, Ladislau Bölöni, Muthucumaru Maheswaran, Albert I. Reuther, James P. Robertson, Mitchell D. Theys, Bin Yao, Debra A. Hensgen, and Richard F. Freund. A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems. *J. Parallel Distrib. Comput.*, 61(6):810–837, 2001.

[22] Tracy D. Braun, Howard Jay Siegel, and Anthony A. Maciejewski. Heterogeneous computing: Goals, methods, and open problems. In Burkhard Monien, Viktor K. Prasanna, and Sriram Vajapeyam, editors, *High Performance Computing - HiPC 2001, 8th International Conference, Hyder-*

abad, India, December, 17-20, 2001, Proceedings, volume 2228 of *Lecture Notes in Computer Science*, pages 307–320. Springer, 2001.

[23] Daniel Brélaz. New methods to color the vertices of a graph. *Commun. ACM*, 22(4):251–256, April 1979.

[24] John Brennan, Ibad Kureshi, and Violeta Holmes. CDES: an approach to HPC workload modelling. In *18th IEEE/ACM International Symposium on Distributed Simulation and Real Time Applications, DS-RT 2014, Toulouse, France, October 1-3, 2014*, pages 47–54. IEEE Computer Society, 2014.

[25] Thomas Bridi, Andrea Bartolini, Michele Lombardi, Michela Milano, and Luca Benini. A constraint programming scheduler for heterogeneous high-performance computing machines. *IEEE Trans. Parallel Distrib. Syst.*, 27(10):2781–2794, 2016.

[26] Jirachai Buddhakulsomsiri and David S. Kim. Priority rule-based heuristic for multi-mode resource-constrained project scheduling problems with resource vacations and activity splitting. *European Journal of Operational Research*, 178(2):374–390, 2007.

[27] Jacques Carlier and Eric Pinson. A practical use of jackson's preemptive schedule for solving the job shop problem. *Annals of Operations Research*, 26(1):269–287, 1990.

[28] Carlo Cavazzoni. EURORA: a european architecture toward exascale. In *Proceedings of the Future HPC Systems - the Challenges of Power-Constrained Performance, FutureHPC@ICS 2012, Venezia, Italy, June 25, 2012*, pages 1:1–1:4. ACM, 2012.

[29] Aftab Ahmed Chandio, Cheng-Zhong Xu, Nikos Tziritas, Kashif Bilal, and Samee Ullah Khan. A comparative study of job scheduling strategies in large-scale parallel computational systems. In *12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications, TrustCom 2013 / 11th IEEE International Symposium on Parallel and Distributed Processing with Applications, ISPA-13 / 12th IEEE International Conference on Ubiquitous Computing and Communications, IUCC-2013, Melbourne, Australia, July 16-18, 2013*, pages 949–957. IEEE Computer Society, 2013.

[30] Steve J. Chapin, Walfredo Cirne, Dror G. Feitelson, James Patton Jones, Scott T. Leutenegger, Uwe Schwiegelshohn, Warren Smith, and David Talby. Benchmarks and standards for the evaluation of parallel job schedulers. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing, IPPS/SPDP'99 Workshop, JSSPP'99, San Juan, Puerto Rico, April 16, 1999, Proceedings*, volume 1659 of *Lecture Notes in Computer Science*, pages 67–90. Springer, 1999.

[31] Xin Chen, Charng-Da Lu, and Karthik Pattabiraman. Predicting job completion times using system logs in supercomputing clusters. In *43rd Annual IEEE/IFIP Conference on Dependable Systems and Networks Workshop, DSN Workshops 2013, Budapest, Hungary, June 24-27, 2013*, pages 1–8. IEEE Computer Society, 2013.

[32] Y. Chen, S. Alspaugh, and R. H. Katz. Design insights for mapreduce from diverse production workloads. Technical report, University of California, Berkeley, 2012.

[33] Su-Hui Chiang, Andrea C. Arpaci-Dusseau, and Mary K. Vernon. The impact of more accurate requested runtimes on production job scheduling performance. In Dror G. Feitelson, Larry Rudolph, and Uwe Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing, 8th International Workshop, JSSPP 2002, Edinburgh, Scotland, UK, July 24, 2002, Revised Papers*, volume 2537 of *Lecture Notes in Computer Science*, pages 103–127. Springer, 2002.

[34] W. Cirne and F. Berman. A comprehensive model of the supercomputer workload. In *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538)*, pages 140–148, Dec 2001.

[35] Edward Grady Coffman and John L Bruno. *Computer and job-shop scheduling theory*. John Wiley & Sons, 1976.

[36] Broderick Crawford, Ricardo Soto, Carlos Castro, and Eric Monfroy. A hyperheuristic approach for dynamic enumeration strategy selection in constraint satisfaction. In José Manuel Ferrández, José Ramón Álvarez Sánchez, Félix de la Paz, and F. Javier Toledo, editors, *New Challenges on Bioinspired Applications - 4th International Work-conference on the Interplay Between Natural and Artificial Computation, IWINAC 2011, La Palma, Canary Islands, Spain, May 30 - June 3, 2011. Proceedings, Part II*, volume 6687 of *Lecture Notes in Computer Science*, pages 295–304. Springer, 2011.

[37] Broderick Crawford, Ricardo Soto, Eric Monfroy, Wenceslao Palma, Carlos Castro, and Fernando Paredes. Parameter tuning of a choice-function based hyperheuristic using particle swarm optimization. *Expert Syst. Appl.*, 40(5):1690–1695, 2013.

[38] Romuald Debruyne and Christian Bessière. Some practicable filtering techniques for the constraint satisfaction problem. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence, IJCAI 97, Nagoya, Japan, August 23-29, 1997, 2 Volumes*, pages 412–417. Morgan Kaufmann, 1997.

[39] Rina Dechter and Judea Pearl. Network-based heuristics for constraint-satisfaction problems. *Artif. Intell.*, 34(1):1–38, 1987.

[40] Pierre-François Dutot, Michael Mercier, Millian Poquet, and Olivier Richard. Batsim: A realistic language-independent resources and jobs management systems simulator. In Narayan Desai and Walfredo Cirne, editors, *Job Scheduling Strategies for Parallel Processing - 19th and 20th International Workshops, JSSPP 2015, Hyderabad, India, May 26, 2015 and JSSPP 2016, Chicago, IL, USA, May 27, 2016, Revised Selected Papers*, volume 10353 of *Lecture Notes in Computer Science*, pages 178–197. Springer, 2016.

[41] Stephen J Ezell and Robert D Atkinson. The vital importance of high-performance computing to us competitiveness. *Information Technology and Innovation Foundation, April*, 28, 2016.

[42] Yuping Fan, Paul Rich, William E. Allcock, Michael E. Papka, and Zhiling Lan. Trade-off between prediction accuracy and underestimation rate in job runtime estimates. In *2017 IEEE International Conference on Cluster Computing, CLUSTER 2017, Honolulu, HI, USA, September 5-8, 2017*, pages 530–540. IEEE Computer Society, 2017.

[43] Dror G. Feitelson. Metrics for parallel job scheduling and their convergence. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing, 7th International Workshop, JSSPP 2001, Cambridge, MA, USA, June 16, 2001, Revised Papers*, volume 2221 of *Lecture Notes in Computer Science*, pages 188–206. Springer, 2001.

[44] Dror G. Feitelson, Larry Rudolph, and Uwe Schwiegelshohn. Parallel job scheduling - A status report. In Dror G. Feitelson, Larry Rudolph, and Uwe Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing, 10th International Workshop, JSSPP 2004, New York, NY, USA, June 13, 2004, Revised Selected Papers*, volume 3277 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2004.

[45] Dror G. Feitelson, Larry Rudolph, Uwe Schwiegelshohn, Kenneth C. Sevcik, and Parkson Wong. Theory and practice in parallel job scheduling. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing, IPPS'97 Workshop, Geneva, Switzerland, April 5, 1997, Proceedings*, volume 1291 of *Lecture Notes in Computer Science*, pages 1–34. Springer, 1997.

[46] Dror G. Feitelson, Dan Tsafrir, and David Krakov. Experience with using the parallel workloads archive. *J. Parallel Distrib. Comput.*, 74(10):2967–2982, 2014.

[47] Dror G. Feitelson and Ahuva Mu'alem Weil. Utilization and predictability in scheduling the IBM SP2 with backfilling. In *12th International Parallel Processing Symposium / 9th Symposium on Parallel and Distributed Processing (IPPS/SPDP '98), March 30 - April 3, 1998, Orlando, Florida, USA, Proceedings*, pages 542–546. IEEE Computer Society, 1998.

[48] Cristian Galleguillos, Zeynep Kiziltan, and Alessio Netti. Accasim: An HPC simulator for workload management. In Esteban E. Mocskos and Sergio Nesmachnow, editors, *High Performance Computing - 4th Latin American Conference, CARLA 2017, Buenos Aires, Argentina, and Colonia del Sacramento, Uruguay, September 20-22, 2017, Revised Selected Papers*, volume 796 of *Communications in Computer and Information Science*, pages 169–184. Springer, 2017.

[49] Cristian Galleguillos, Zeynep Kiziltan, Alessio Netti, and Ricardo Soto. Accasim: a customizable workload management simulator for job dispatching research in HPC systems. *Cluster Computing*, 23(1):107–122, 2020.

[50] Cristian Galleguillos, Zeynep Kiziltan, Alina Sîrbu, and Özalp Babaoglu. Constraint programming-based job dispatching for modern HPC applications. In Thomas Schiex and Simon de Givry, editors, *Principles and Practice of Constraint Programming - 25th International Conference, CP 2019, Stamford, CT, USA, September 30 - October 4, 2019, Proceedings*, volume 11802 of *Lecture Notes in Computer Science*, pages 438–455. Springer, 2019.

[51] Cristian Galleguillos, Alina Sîrbu, Zeynep Kiziltan, Özalp Babaoglu, Andrea Borghesi, and Thomas Bridi. Data-driven job dispatching in HPC systems. In Giuseppe Nicosia, Panos M. Pardalos, Giovanni Giuffrida, and Renato Umeton, editors, *Machine Learning, Optimization, and Big Data - Third International Conference, MOD 2017, Volterra, Italy, September 14-17, 2017, Revised Selected Papers*, volume 10710 of *Lecture Notes in Computer Science*, pages 449–461. Springer, 2017.

[52] Éric Gaussier, David Glesser, Valentin Reis, and Denis Trystram. Improving backfilling by using machine learning to predict running times. In Jackie Kern and Jeffrey S. Vetter, editors, *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2015, Austin, TX, USA, November 15-20, 2015*, pages 64:1–64:10. ACM, 2015.

[53] P. A. Geelen. Dual viewpoint heuristics for binary constraint satisfaction problems. In Bernd Neumann, editor, *10th European Conference on Artificial Intelligence, ECAI 92, Vienna, Austria, August 3-7, 1992. Proceedings.*, pages 31–35. John Wiley and Sons, 1992.

[54] Matthew L. Ginsberg, Michael Frank, Michael P. Halpin, and Mark C. Torrance. Search lessons learned from crossword puzzles. In Howard E. Shrobe, Thomas G. Dietterich, and William R. Swartout, editors, *Proceedings of the 8th National Conference on Artificial Intelligence. Boston, Massachusetts, USA, July 29 - August 3, 1990, 2 Volumes.*, pages 210–215. AAAI Press / The MIT Press, 1990.

[55] Daniel Godard, Philippe Laborie, and Wim Nuijten. Randomized large neighborhood search for cumulative scheduling. In Susanne Biundo, Karen L. Myers, and Kanna Rajan, editors, *Proceedings of the Fifteenth International Conference on Automated Planning and Scheduling (ICAPS 2005), June 5-10 2005, Monterey, California, USA*, pages 81–89. AAAI, 2005.

[56] Solomon W. Golomb and Leonard D. Baumert. Backtrack programming. *J. ACM*, 12(4):516–524, 1965.

[57] Carla P. Gomes, Bart Selman, and Henry A. Kautz. Boosting combinatorial search through randomization. In Jack Mostow and Chuck Rich, editors, *Proceedings of the Fifteenth National Conference on Artificial Intelligence and Tenth Innovative Applications of Artificial Intelligence Conference, AAAI 98, IAAI 98, July 26-30, 1998, Madison, Wisconsin, USA.*, pages 431–437. AAAI Press / The MIT Press, 1998.

[58] César Gómez-Martín, Miguel A. Vega-Rodríguez, and José Luis González Sánchez. Performance and energy aware scheduling simulator for HPC: evaluating different resource selection methods. *Concurrency and Computation: Practice and Experience*, 27(17):5436–5459, 2015.

[59] Youssef Hamadi, Eric Monfroy, and Frédéric Saubion, editors. *Autonomous Search*. Springer, 2012.

[60] Per Brinch Hansen. An analysis of response ratio scheduling. In Charles V. Freiman, John E. Griffith, and Jack L. Rosenfeld, editors, *Information Processing, Proceedings of IFIP Congress 1971, Volume 1 - Foundations and Systems, Ljubljana, Yugoslavia, August 23-28, 1971*, pages 479–484. North-Holland, 1971.

[61] Robert M. Haralick and Gordon L. Elliott. Increasing tree search efficiency for constraint satisfaction problems. *Artif. Intell.*, 14(3):263–313, 1980.

[62] R. Haupt. A survey of priority rule-based scheduling. *Operations-Research-Spektrum*, 11(1):3–16, Mar 1989.

[63] Robert L. Henderson. Job scheduling under the portable batch system. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing, IPPS'95 Workshop, Santa Barbara, CA, USA, April 25, 1995, Proceedings*, volume 949 of *Lecture Notes in Computer Science*, pages 279–294. Springer, 1995.

[64] Pascal Van Hentenryck, Vijay A. Saraswat, and Yves Deville. Design, implementation, and evaluation of the constraint language cc(fd). *J. Log. Program.*, 37(1-3):139–164, 1998.

[65] Matthias Hovestadt, Odej Kao, Axel Keller, and Achim Streit. Scheduling in HPC resource management systems: Queuing vs. planning. In Dror G.

Feitelson, Larry Rudolph, and Uwe Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing, 9th International Workshop, JSSPP 2003, Seattle, WA, USA, June 24, 2003, Revised Papers*, volume 2862 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2003.

[66] William B. Hurst, Srini Ramaswamy, R. B. Lenin, and D. Hoffman. Modeling and simulation of hpc systems through job scheduling analysis. In *Conference on Applied Research in Information Technology.* Acxiom Laboratory of Applied Research, 2010.

[67] Michael R. Wyatt II, Stephen Herbein, Todd Gamblin, Adam Moody, Dong H. Ahn, and Michela Taufer. PRIONN: predicting runtime and IO using neural networks. In *Proceedings of the 47th International Conference on Parallel Processing, ICPP 2018, Eugene, OR, USA, August 13-16, 2018*, pages 46:1–46:12. ACM, 2018.

[68] David B. Jackson, Quinn Snell, and Mark J. Clement. Core algorithms of the maui scheduler. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing, 7th International Workshop, JSSPP 2001, Cambridge, MA, USA, June 16, 2001, Revised Papers*, volume 2221 of *Lecture Notes in Computer Science*, pages 87–102. Springer, 2001.

[69] Nikhil Jain, Abhinav Bhatele, Sam White, Todd Gamblin, and Laxmikant V. Kalé. Evaluating HPC networks via simulation of parallel workloads. In John West and Cherri M. Pancake, editors, *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2016, Salt Lake City, UT, USA, November 13-18, 2016*, pages 154–165. IEEE Computer Society, 2016.

[70] Berit Johannes. Scheduling parallel jobs to minimize the makespan. *J. Scheduling*, 9(5):433–452, 2006.

[71] Mala Kalra and Sarbjeet Singh. A review of metaheuristic scheduling techniques in cloud computing. *Egyptian Informatics Journal*, 16(3):275 – 295, 2015.

[72] Dalibor Klusacek and Hana Rudova. Improving qos in computational grids through schedule-based approach. In *Scheduling and Planning Applications Workshop at the Eighteenth International Conference on Automated Planning and Scheduling (ICAPS 2008), Sydney, Australia*, 2008.

[73] Dalibor Klusácek and Hana Rudová. Alea 2: job scheduling simulator. In L. Felipe Perrone, Giovanni Stea, Jason Liu, Adelinde M. Uhrmacher, and Manuel Villén-Altamirano, editors, *3rd International Conference on Simulation Tools and Techniques, SIMUTools '10, Malaga, Spain - March 16 - 18, 2010*, page 61. ICST/ACM, 2010.

[74] P. Laborie and D. Godard. Self-adapting large neighborhood search:application to single-mode scheduling problems. In *Proc. of 3rd Multidisciplinary International Conference on Scheduling : Theory and Applications, MISTA 2007*, pages 276–284, 2007.

[75] Philippe Laborie and Jerome Rogerie. Reasoning with conditional time-intervals. In David Wilson and H. Chad Lane, editors, *Proceedings of the Twenty-First International Florida Artificial Intelligence Research Society Conference, May 15-17, 2008, Coconut Grove, Florida, USA*, pages 555–560. AAAI Press, 2008.

[76] Philippe Laborie, Jerome Rogerie, Paul Shaw, and Petr Vilím. Reasoning with conditional time-intervals. part II: an algebraical model for resources. In H. Chad Lane and Hans W. Guesgen, editors, *Proceedings of the Twenty-Second International Florida Artificial Intelligence Research Society Conference, May 19-21, 2009, Sanibel Island, Florida, USA*. AAAI Press, 2009.

[77] Philippe Laborie, Jerome Rogerie, Paul Shaw, and Petr Vilím. IBM ILOG CP optimizer for scheduling - 20+ years of scheduling with constraints at IBM/ILOG. *Constraints*, 23(2):210–250, 2018.

[78] Abdelkader Lahrichi. Scheduling-the notions of hump, compulsory parts and their use in cumulative problems. *COMPTES RENDUS DE L ACADEMIE DES SCIENCES SERIE I-MATHEMATIQUE*, 294(6):209–211, 1982.

[79] Cynthia Bailey Lee, Yael Schwartzman, Jennifer Hardy, and Allan Snavely. Are user runtime estimates inherently inaccurate? In Dror G. Feitelson, Larry Rudolph, and Uwe Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing, 10th International Workshop, JSSPP 2004, New York, NY, USA, June 13, 2004, Revised Selected Papers*, volume 3277 of *Lecture Notes in Computer Science*, pages 253–263. Springer, 2004.

[80] Jérôme Lelong, Valentin Reis, and Denis Trystram. Tuning easy-backfilling queues. In Dalibor Klusácek, Walfredo Cirne, and Narayan Desai, editors, *Job Scheduling Strategies for Parallel Processing - 21st International Workshop, JSSPP 2017, Orlando, FL, USA, June 2, 2017, Revised Selected Papers*, volume 10773 of *Lecture Notes in Computer Science*, pages 43–61. Springer, 2017.

[81] Yawei Li, Prashasta Gujrati, Zhiling Lan, and Xian-He Sun. Fault-driven re-scheduling for improving system-level fault resilience. In *2007 International Conference on Parallel Processing (ICPP 2007), September 10-14, 2007, Xi-An, China*, page 39. IEEE Computer Society, 2007.

[82] David A. Lifka. The ANL/IBM SP scheduling system. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel*

Processing, IPPS'95 Workshop, Santa Barbara, CA, USA, April 25, 1995, Proceedings, volume 949 of *Lecture Notes in Computer Science*, pages 295–303. Springer, 1995.

[83] Feng Liu and Jon B. Weissman. Elastic job bundling: an adaptive resource request strategy for large-scale parallel applications. In Jackie Kern and Jeffrey S. Vetter, editors, *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2015, Austin, TX, USA, November 15-20, 2015*, pages 33:1–33:12. ACM, 2015.

[84] Michele Lombardi and Michela Milano. A precedence constraint posting approach for the RCPSP with time lags and variable durations. In Ian P. Gent, editor, *Principles and Practice of Constraint Programming - CP 2009, 15th International Conference, CP 2009, Lisbon, Portugal, September 20-24, 2009, Proceedings*, volume 5732 of *Lecture Notes in Computer Science*, pages 569–583. Springer, 2009.

[85] Uri Lublin and Dror G. Feitelson. The workload on parallel supercomputers: modeling the characteristics of rigid jobs. *J. Parallel Distrib. Comput.*, 63(11):1105–1122, 2003.

[86] Michael Luby, Alistair Sinclair, and David Zuckerman. Optimal speedup of las vegas algorithms. *Inf. Process. Lett.*, 47(4):173–180, 1993.

[87] Alejandro Lucero. Simulation of batch scheduling using real production-ready software tools. In *Proc. of IBERGRID'11*, pages 345–356. Netbiblo, 2011.

[88] Andréa M. Matsunaga and José A. B. Fortes. On the use of machine learning to predict the time and resources consumed by applications. In *10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing, CCGrid 2010, 17-20 May 2010, Melbourne, Victoria, Australia*, pages 495–504. IEEE Computer Society, 2010.

[89] M. Milano, editor. *Constraints: Anniversary Special Issue*, volume 23. Springer, 2018.

[90] Nader Mohamed and Jameela Al-Jaroodi. Real-time big data analytics: Applications and challenges. In *International Conference on High Performance Computing & Simulation, HPCS 2014, Bologna, Italy, 21-25 July, 2014*, pages 305–310. IEEE, 2014.

[91] Misbah Mubarak, Christopher D. Carothers, Robert B. Ross, and Philip H. Carns. Enabling parallel simulation of large-scale HPC network systems. *IEEE Trans. Parallel Distrib. Syst.*, 28(1):87–100, 2017.

[92] Prakash Murali and Sathish Vadhiyar. Metascheduling of HPC jobs in day-ahead electricity markets. *IEEE Trans. Parallel Distrib. Syst.*, 29(3):614–627, 2018.

[93] Alessio Netti, Cristian Galleguillos, Zeynep Kiziltan, Alina Sîrbu, and Özalp Babaoglu. Heterogeneity-aware resource allocation in HPC systems. In Rio Yokota, Michèle Weiland, David E. Keyes, and Carsten Trinitis, editors, *High Performance Computing - 33rd International Conference, ISC High Performance 2018, Frankfurt, Germany, June 24-28, 2018, Proceedings*, volume 10876 of *Lecture Notes in Computer Science*, pages 3–21. Springer, 2018.

[94] Jorji Nonaka, Naohisa Sakamoto, Takashi Shimizu, Masahiro Fujita, Kenji Ono, and Koji Koyamada. Distributed particle-based rendering framework for large data visualization on HPC environments. In *2017 International Conference on High Performance Computing & Simulation, HPCS 2017, Genoa, Italy, July 17-21, 2017*, pages 300–307. IEEE, 2017.

[95] Alberto Nuñez, Javier Fernández, José Daniel García, Félix García, and Jesús Carretero. New techniques for simulating high performance MPI applications on large storage networks. *The Journal of Supercomputing*, 51(1):40–57, 2010.

[96] C. Le Pape, P. Couronne, D. Vergamini, and V. Gosselin. Time-versus-capacity compromises in project scheduling. *AISB Quartetly*, pages 19–31, 1995.

[97] Claude Le Pape, Philippe Couronné, Didier Vergamini, and Vincent Gosselin. Time-versus-capacity compromises in project scheduling, 1994.

[98] Laurent Péridy and David Rivreau. An o (n log n) stable algorithm for immediate selections adjustments. In *Multidisciplinary Scheduling: Theory and Applications*, pages 205–222. Springer, 2005.

[99] Michael Pinedo. *Scheduling*, volume 29. Springer, 2012.

[100] Jean-Charles Régin. Minimization of the number of breaks in sports scheduling problems using constraint programming. In Eugene C. Freuder and Richard J. Wallace, editors, *Constraint Programming and Large Scale Discrete Optimization, Proceedings of a DIMACS Workshop, Princeton, New Jersey, USA, September 14-17, 1998*, volume 57 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 115–130. DIMACS/AMS, 1998.

[101] Charles Reiss, Alexey Tumanov, Gregory R. Ganger, Randy H. Katz, and Michael A. Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In Michael J. Carey and Steven Hand, editors, *ACM Symposium on Cloud Computing, SOCC '12, San Jose, CA, USA, October 14-17, 2012*, page 7. ACM, 2012.

[102] Albert Reuther, Chansup Byun, William Arcand, David Bestor, Bill Bergeron, Matthew Hubbell, Michael Jones, Peter Michaleas, Andrew Prout, Antonio Rosa, and Jeremy Kepner. Scalable system scheduling for HPC and big data. *J. Parallel Distrib. Comput.*, 111:76–92, 2018.

[103] Gonzalo P. Rodrigo, Erik Elmroth, Per-Olov Östberg, and Lavanya Ramakrishnan. Scsf: A scheduling simulation framework. In Dalibor Klusácek, Walfredo Cirne, and Narayan Desai, editors, *Job Scheduling Strategies for Parallel Processing - 21st International Workshop, JSSPP 2017, Orlando, FL, USA, June 2, 2017, Revised Selected Papers*, volume 10773 of *Lecture Notes in Computer Science*, pages 152–173. Springer, 2017.

[104] Joaquín Rodriguez. A constraint programming model for real-time train scheduling at junctions. *Transportation Research Part B: Methodological*, 41(2):231–245, 2007.

[105] Francesca Rossi, Peter van Beek, and Toby Walsh, editors. *Handbook of Constraint Programming*, volume 2 of *Foundations of Artificial Intelligence*. Elsevier, 2006.

[106] C. Rückemann. Using parallel multicore and HPC systems for dynamical visualisation. In *2009 International Conference on Advanced Geographic Information Systems Web Services*, pages 13–18, 2009.

[107] Paul Shaw. Using constraint programming and local search methods to solve vehicle routing problems. In Michael J. Maher and Jean-Francois Puget, editors, *Principles and Practice of Constraint Programming - CP98, 4th International Conference, Pisa, Italy, October 26-30, 1998, Proceedings*, volume 1520 of *Lecture Notes in Computer Science*, pages 417–431. Springer, 1998.

[108] Edi Shmueli and Dror G. Feitelson. Backfilling with lookahead to optimize the performance of parallel job scheduling. In Dror G. Feitelson, Larry Rudolph, and Uwe Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing, 9th International Workshop, JSSPP 2003, Seattle, WA, USA, June 24, 2003, Revised Papers*, volume 2862 of *Lecture Notes in Computer Science*, pages 228–251. Springer, 2003.

[109] Hayk Shoukourian, Torsten Wilde, Axel Auweter, and Arndt Bode. Predicting the energy and power consumption of strong and weak scaling hpc applications. *Supercomput. Front. Innov.: Int. J.*, 1(2):20–41, July 2014.

[110] Helmut Simonis. Sudoku as a constraint problem. In *CP Workshop on modeling and reformulating Constraint Satisfaction Problems*, volume 12, pages 13–27. Citeseer, 2005.

[111] Poonam Singh, Maitreyee Dutta, and Naveen Aggarwal. A review of task scheduling based on meta-heuristics approach in cloud computing. *Knowl. Inf. Syst.*, 52(1):1–51, 2017.

[112] Alina Sîrbu and Özalp Babaoglu. A holistic approach to log data analysis in high-performance computing systems: The case of IBM blue gene/q. In Sascha Hunold, Alexandru Costan, Domingo Giménez, Alexandru Iosup,

Laura Ricci, María Engracia Gómez Requena, Vittorio Scarano, Ana Lucia Varbanescu, Stephen L. Scott, Stefan Lankes, Josef Weidendorfer, and Michael Alexander, editors, *Euro-Par 2015: Parallel Processing Workshops - Euro-Par 2015 International Workshops, Vienna, Austria, August 24-25, 2015, Revised Selected Papers*, volume 9523 of *Lecture Notes in Computer Science*, pages 631–643. Springer, 2015.

[113] SLURM. SLURM workload manager, 2019.

[114] Shane Snyder, Philip H. Carns, Robert Latham, Misbah Mubarak, Robert B. Ross, Christopher D. Carothers, Babak Behzad, Huong Vu Thanh Luu, Surendra Byna, and Prabhat. Techniques for modeling large-scale HPC I/O workloads. In Stephen A. Jarvis, Steven A. Wright, and Simon D. Hammond, editors, *Proceedings of the 6th International Workshop on Performance Modeling, Benchmarking, and Simulation of High Performance Computing Systems, PMBS 2015, Austin, Texas, USA, November 15, 2015*, pages 5:1–5:11. ACM, 2015.

[115] Ricardo Soto, Broderick Crawford, Cristian Galleguillos, Fernando Paredes, and Enrique Norero. A hybrid alldifferent-tabu search algorithm for solving sudoku puzzles. *Comp. Int. and Neurosc.*, 2015:286354:1–286354:10, 2015.

[116] Ricardo Soto, Broderick Crawford, Rodrigo Olivares, Cristian Galleguillos, Carlos Castro, Franklin Johnson, Fernando Paredes, and Enrique Norero. Using autonomous search for solving constraint satisfaction problems via new modern approaches. *Swarm and Evolutionary Computation*, 30:64–77, 2016.

[117] Ricardo Soto, Broderick Crawford, Wenceslao Palma, Karin Galleguillos, Carlos Castro, Eric Monfroy, Franklin Johnson, and Fernando Paredes. Boosting autonomous search for csps via skylines. *Inf. Sci.*, 308:38–48, 2015.

[118] Mehmet Soysal, Marco Berghoff, and Achim Streit. Analysis of job metadata for enhanced wall time prediction. In Dalibor Klusácek, Walfredo Cirne, and Narayan Desai, editors, *Job Scheduling Strategies for Parallel Processing - 22nd International Workshop, JSSPP 2018, Vancouver, BC, Canada, May 25, 2018, Revised Selected Papers*, volume 11332 of *Lecture Notes in Computer Science*, pages 1–14. Springer, 2018.

[119] Srividya Srinivasan, Rajkumar Kettimuthu, Vijay Subramani, and P. Sadayappan. Characterization of backfilling strategies for parallel job scheduling. In *31st International Conference on Parallel Processing Workshops (ICPP 2002 Workshops), 20-23 August 2002, Vancouver, BC, Canada*, pages 514–522. IEEE Computer Society, 2002.

[120] Srividya Srinivasan, Rajkumar Kettimuthu, Vijay Subramani, and P. Sadayappan. Selective reservation strategies for backfill job scheduling. In

Dror G. Feitelson, Larry Rudolph, and Uwe Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing, 8th International Workshop, JSSPP 2002, Edinburgh, Scotland, UK, July 24, 2002, Revised Papers*, volume 2537 of *Lecture Notes in Computer Science*, pages 55–71. Springer, 2002.

[121] Ingo Steinwart and Andreas Christmann. *Support Vector Machines*. Information science and statistics. Springer, 2008.

[122] Trofinoff Stephen and Massimo Benini. Using and modifying the bsc slurm workload simulator. Technical report, Slurm User Group Meeting, 2015.

[123] Curtis Storlie, Joe Sexton, Scott Pakin, Michael Lang, Brian Reich, and William Rust. Modeling and predicting power consumption of high performance computing jobs. *preprint arXiv:1412.5247*, 2014.

[124] Achim Streit. Enhancements to the decision process of the self-tuning dynp scheduler. In Dror G. Feitelson, Larry Rudolph, and Uwe Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing, 10th International Workshop, JSSPP 2004, New York, NY, USA, June 13, 2004, Revised Selected Papers*, volume 3277 of *Lecture Notes in Computer Science*, pages 63–80. Springer, 2004.

[125] Wolfgang Süß, Wilfried Jakob, Alexander Quinte, and Karl-Uwe Stucky. GORBA: A global optimising resource broker embedded in a grid resource management system. In S. Q. Zheng, editor, *International Conference on Parallel and Distributed Computing Systems, PDCS 2005, November 14-16, 2005, Phoenix, AZ, USA*, pages 19–24. IASTED/ACTA Press, 2005.

[126] David Talby and Dror G. Feitelson. Supporting priorities and improving utilization of the IBM SP scheduler using slack-based backfilling. In *13th International Parallel Processing Symposium / 10th Symposium on Parallel and Distributed Processing (IPPS / SPDP '99), 12-16 April 1999, San Juan, Puerto Rico, Proceedings*, pages 513–517. IEEE Computer Society, 1999.

[127] Qinghui Tang, Sandeep K. S. Gupta, and Georgios Varsamopoulos. Energy-efficient thermal-aware task scheduling for homogeneous high-performance computing data centers: A cyber-physical approach. *IEEE Trans. Parallel Distrib. Syst.*, 19(11):1458–1472, 2008.

[128] Wei Tang, Narayan Desai, Daniel Buettner, and Zhiling Lan. Analyzing and adjusting user runtime estimates to improve job scheduling on the blue gene/p. In *24th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2010, Atlanta, Georgia, USA, 19-23 April 2010 - Conference Proceedings*, pages 1–11. IEEE, 2010.

[129] Dan Tsafrir, Yoav Etsion, and Dror G. Feitelson. Backfilling using system-generated predictions rather than user runtime estimates. *IEEE Trans. Parallel Distrib. Syst.*, 18(6):789–803, 2007.

[130] Peter van Beek. Backtracking search algorithms. In Francesca Rossi, Peter van Beek, and Toby Walsh, editors, *Handbook of Constraint Programming*, volume 2 of *Foundations of Artificial Intelligence*, pages 85–134. Elsevier, 2006.

[131] P. Van Hentenryck and M. Milano, editors. *Hybrid optimization: the ten years of CPAIOR*, volume 45. Springer Science & Business Media, 2010.

[132] Petr Vilím. O (nlogn) filtering algorithms for unary resource constraint. In *International Conference on Integration of Artificial Intelligence (AI) and Operations Research (OR) Techniques in Constraint Programming*, pages 335–347. Springer, 2004.

[133] Fabien Vivodtzev and Isabelle Bertron. Remote visualization of large scale fast dynamic simulations in a HPC context. In Hank Childs, Renato Pajarola, and Venkatram Vishwanath, editors, *4th IEEE Symposium on Large Data Analysis and Visualization, LDAV 2014, Paris, France, November 9-10, 2014*, pages 121–122. IEEE Computer Society, 2014.

[134] Ahuva Mu'alem Weil and Dror G. Feitelson. Utilization, predictability, workloads, and user runtime estimates in scheduling the IBM SP2 with backfilling. *IEEE Trans. Parallel Distrib. Syst.*, 12(6):529–543, 2001.

[135] Georges Weil, Kamel Heus, Patrice Francois, and Marc Poujade. Constraint programming for nurse scheduling. *IEEE Engineering in medicine and biology magazine*, 14(4):417–422, 1995.

[136] Adam K. L. Wong and Andrzej M. Goscinski. Evaluating the easy-backfill job scheduling of static workloads on clusters. In *Proceedings of the 2007 IEEE International Conference on Cluster Computing, 17-20 September 2007, Austin, Texas, USA*, pages 64–73. IEEE Computer Society, 2007.

[137] Jörg Würtz and Tobias Müller. Constructive disjunction revisited. In Günther Görz and Steffen Hölldobler, editors, *KI-96: Advances in Artificial Intelligence, 20th Annual German Conference on Artificial Intelligence, Dresden, Germany, September 17-19, 1996, Proceedings*, volume 1137 of *Lecture Notes in Computer Science*, pages 377–386. Springer, 1996.

[138] Zhou Zhou, Zhiling Lan, Wei Tang, and Narayan Desai. Reducing energy costs for IBM blue gene/p via power-aware job scheduling. In Narayan Desai and Walfredo Cirne, editors, *Job Scheduling Strategies for Parallel Processing - 17th International Workshop, JSSPP 2013, Boston, MA, USA, May 24, 2013 Revised Selected Papers*, volume 8429 of *Lecture Notes in Computer Science*, pages 96–115. Springer, 2013.