Learning about simple heuristics for online parallel job scheduling

Danilo Carastan dos Santos

Communauté UNIVERSITÉ Grenoble Alpes

UFABC

THÈSE

Pour obtenir le grade de

DOCTEUR DE LA COMMUNAUTÉ UNIVERSITÉ GRENOBLE ALPES

préparée dans le cadre d'une cotutelle entre la Communauté Université Grenoble Alpes et la Fundação Universidade Federal do ABC

Spécialité : **Informatique**

Arrêté ministériel : 6 janvier 2005 - 25 mai 2016

Présentée par

Danilo CARASTAN DOS SANTOS

Thèse dirigée par **Denis TRYSTRAM**
et codirigée par **Raphael Y. DE CAMARGO**

préparée au sein des laboratoires **Laboratoire d'Informatique de Grenoble** et **Laboratório de Computação Paralela e Distribuida**
dans les Écoles Doctorales **MSTII** et **Programa de Pós-Graduação em Ciência da Computação**

Apprentissage sur Heuristiques Simples pour l'Ordonnancement *Online* de Tâches Parallèles

Learning About Simple Heuristics for Online Parallel Job Scheduling

Thèse soutenue publiquement le **27 novembre 2019**,
devant le jury composé de :

Martin SCHULZ
Professeur d'Université, Leibniz Supercomputing Centre, Technische Universität München, Allemagne, Rapporteur

Alfredo GOLDMAN VEL LEJBMAN
Professeur Associé, IME-USP, Universidade de São Paulo, Brésil, Président

Sarita BRUSCHI
Professeure Assistante, ICMC-USP, Universidade de São Paulo, Brésil, Examinatrice

Éric GAUSSIER
Directeur de Laboratoire, LIG, Univ. Grenoble Alpes, France, Examinateur

Raphael Y. DE CAMARGO
Professeur d'Université, CMCC, Universidade Federal do ABC, Brésil, Co-Directeur de thèse

Denis TRYSTRAM
Professeur d'Université, LIG, Univ. Grenoble Alpes, France, Directeur de thèse

Aos meus pais, Elena e Claudio.

Absque sudore et labore nullum opus perfectum est.

— **Schrevelius**

Remerciements (Acknowledgments)

There is not a single piece of work whose realization depends only on its author. This thesis is no exception to this rule. There has been many people who contributed directly or indirectly to the realization of this thesis. This section is dedicated to show my appreciation to all of these people. I hope that I remember all of them.

The first and perhaps most important persons are my parents, Elena and Claudio, whose support has been critical to my development on both professional and personal levels. I would have never reached to where I am without them.

I would also like to thank the former Brazillian government, from 2003 to 2010, which had as its leader the former president Luiz Ignácio “Lula” da Silva. It was due to their efforts that had enabled the conception of the Universidade Federal do ABC (UFABC), which was for me the entry point for an enormous development, on both intellectual and economical levels. UFABC is being a key agent to close the abysmal social and economical gap that exists in Brazil, specially in São Paulo, not only for me but for many thousands of other people.

I would like to thank as well my former Master’s advisor, Luiz Rozante, who had trusted me in 2011 that I could perform a good work in research. By the time I had nothing noteworthy to show besides “a mediocre writing skill”. Many years after this trust vote we managed to have one of the ten best Brazillian Master’s thesis in Computer Science of 2015, mainly due to the many skills that I had developed and honed during his supervision.

I would like to thank as well my Brazillian PhD advisor, Raphael Camargo, for all of his support and contribution during the thesis. It is being a pleasure to work with

such a competent, inspiring, and motivating person that Raphael is. There has been countless meetings that pumped me to work on a certain subject, mainly due to Raphael's motivation. Raphael was a key person on all of the contributions of this thesis. All of the thesis award nominations and winnings would not happen without his support and contribution.

I would like to thank a dear friend of mine, Alfredo Goldman, who gave me the opportunity to go to Grenoble for the first time. This visit to Grenoble is being a life-changer for me in many levels.

I would also like to thank my French PhD advisor, Denis Trystram, who accepted me as one of his PhD students. His experience and support has helped me to hone many of my scientific skills. Without a doubt Denis was equally a key person that enabled the conception of this thesis.

I also appreciate the help of my dear friend and colleague Salah Zrigui, who has been a key work partner during the development of this thesis, specially at its second part. I look forward to have more collaborations with him in the future.

I also like to thank all of my friends and colleagues that I have the pleasure to meet at LIG and in Grenoble. You guys have helped me more than you realize.

At last but not least, I would also like to thank Universidade Federal do ABC, Université Grenoble-Alpes, CAPES, FAPESP, Inria, CNRS, and Grenoble INP for all of the financial support that enabled the realization of this thesis.

Abstract / Résumé

Abstract

High-Performance Computing (HPC) platforms are growing in size and complexity. In an adversarial manner, the power demand of such platforms has rapidly grown as well, and current top supercomputers require power at the scale of an entire power plant. In an effort to make a more responsible usage of such power, researchers are devoting a great amount of effort to devise algorithms and techniques to improve different aspects of performance such as scheduling and resource management. But HPC platform maintainers are still reluctant to deploy state of the art scheduling methods and most of them revert to simple heuristics such as EASY Backfilling, which is based in a naive First-Come-First-Served (FCFS) ordering. Newer methods are often complex and obscure, and the simplicity and transparency of EASY Backfilling are too important to sacrifice.

At a first moment we explored Machine Learning (ML) techniques to learn on-line parallel job scheduling heuristics. Using simulations and a workload generation model, we could determine the characteristics of HPC applications (jobs) that lead to a reduction in the mean slowdown of jobs in an execution queue. Modeling these characteristics using a nonlinear function and applying this function to select the next job to execute in a queue improved the mean task slowdown in synthetic workloads. When applied to real workload traces from highly different machines, these functions still resulted in performance improvements, attesting the generalization capability of the obtained heuristics.

At a second moment, using simulations and workload traces from several real HPC platforms, we performed a thorough analysis of the cumulative results of four simple scheduling heuristics (including EASY Backfilling). We also evaluated effects such as the relationship between job size and slowdown, the distribution of slowdown values, and the number of backfilled jobs, for each HPC platform and scheduling policy. We show experimental evidence that one can only gain by replacing EASY Backfilling with the Smallest estimated Area First (SAF) policy with backfilling, as it offers improvements in performance by up to 80% in the slowdown metric while maintaining the simplicity and the transparency of EASY. SAF reduces the number of jobs with large slowdowns and the inclusion of a simple thresholding mechanism guarantees that no starvation occurs.

Overall we achieved the following remarks: (i) simple and efficient scheduling heuristics in the form of a nonlinear function of the jobs characteristics can be learned automatically, though whether the reasoning behind their scheduling decisions is clear or not can be up to argument. (ii) The area (processing time estimate multiplied by the number of processors) of the jobs seems to be a quite important property for good parallel job scheduling heuristics, since many of the heuristics (notably SAF) that achieved good performances have the job's area as input. (iii) The backfilling mechanism seems to always help in increasing performance, though it does not outperform a better sorting of the jobs waiting queue, such as the sorting performed by SAF.

Résumé

Les plate-formes de Calcul Haute Performance (de l'Anglais *High Performance Computing*, HPC) augmentent en taille et en complexité. De manière contradictoire, la demande en énergie de telles plates-formes a également rapidement augmenté. Les supercalculateurs actuels ont besoin d'une puissance équivalente à celle de toute une centrale d'énergie. Dans le but de faire un usage plus responsable de ce puissance de calcul, les chercheurs consacrent beaucoup d'efforts à la conception d'algorithmes et de techniques permettant d'améliorer différents aspects de performance, tels que

l'ordonnancement et la gestion des ressources. Cependant, les responsables des plate-formes HPC hésitent encore à déployer des méthodes d'ordonnancement à la fine pointe de la technologie et la plupart d'entre eux recourent à des méthodes heuristiques simples, telles que l'EASY Backfilling, qui repose sur un tri naïf premier arrivé, premier servi (de l'Anglais *First-Come-First-Served*, FCFS). Les nouvelles méthodes sont souvent complexes et obscures, et la simplicité et la transparence de l'EASY Backfilling sont trop importantes pour être sacrifiées.

Dans un premier temps, nous explorons les techniques d'Apprentissage Automatique (de l'Anglais *Machine Learning*, ML) pour apprendre des méthodes heuristiques d'ordonnancement *online* de tâches parallèles. À l'aide de simulations et d'un modèle de génération de charge de travail, nous avons pu déterminer les caractéristiques des applications HPC (tâches) qui contribuent pour une réduction du ralentissement moyen des tâches dans une file d'attente d'exécution. La modélisation de ces caractéristiques par une fonction non linéaire et l'application de cette fonction pour sélectionner la prochaine tâche à exécuter dans une file d'attente ont amélioré le ralentissement moyen des tâches dans les charges de travail synthétiques. Appliquées à des traces de charges de travail réelles de plate-formes HPC très différents, ces fonctions ont néanmoins permis d'améliorer les performances, attestant de la capacité de généralisation des heuristiques obtenues.

Dans un deuxième temps, à l'aide de simulations et de traces de charge de travail de plusieurs plates-formes HPC réelles, nous avons effectué une analyse approfondie des résultats cumulés de quatre heuristiques simples d'ordonnancement (y compris l'EASY Backfilling). Nous avons également évalué des autres effets tels que la relation entre la taille des tâches et leur ralentissement, la distribution des valeurs de ralentissement et le nombre de tâches mises en calcul par *backfilling*, par chaque plate-forme HPC et politique d'ordonnancement. Nous démontrons de manière expérimentale que l'on ne peut que gagner en remplaçant l'EASY Backfilling par la stratégie SAF (de l'Anglais *Smallest estimated Area First*) aidée par *backfilling*, car elle offre une amélioration des performances allant jusqu'à 80% dans la métrique de ralentissement, tout en maintenant la simplicité et la transparence d'EASY Backfilling. La SAF

réduit le nombre de tâches à hautes valeurs de ralentissement et, par l'inclusion d'un mécanisme de seuillage simple, nous garantissons l'absence d'inanition de tâches.

Dans l'ensemble, nous avons obtenu les remarques suivantes : (i) des heuristiques simples et efficaces sous la forme d'une fonction non linéaire des caractéristiques des tâches peuvent être apprises automatiquement, bien qu'il soit subjectif de conclure si le raisonnement qui sous-tend les décisions d'ordonnancement de ces heuristiques est clair ou non. (ii) La zone (l'estimation du temps d'exécution multipliée par le nombre de processeurs) des tâches semble être une propriété assez importante pour une bonne heuristique d'ordonnancement des tâches parallèles, car un bon nombre d'heuristiques (notamment la SAF) qui ont obtenu de bonnes performances ont la zone de la tâche comme entrée (iii) Le mécanisme de *backfilling* semble toujours contribuer à améliorer les performances, bien que cela ne remédie pas à un meilleur tri de la file d'attente de tâches, tel que celui effectué par SAF.

List of Figures

List of Tables

Contents

1 Introduction

High Performance Computing (HPC) is becoming more important and crucial for many research and industrial applications. The ever increasing computational power is enabling advancements that would not be feasible without the advent of such computing power. A few examples are earthquake simulation with low frequency rate and meters of resolution [36], simulation of quantum phenomena that cannot be fully resolved experimentally such as neutron lifetime [4], and advanced Artificial Intelligence (AI) models with superhuman performance in complex games such as Chess and Go [69].

With all of these advances allowed by HPC, the pursuit of more computing power grows steadily. The Top500 [76] site maintains a ranking of the most powerful supercomputers and this ranking is updated twice a year. At the time of writing this thesis, the most recent list available (November 2018) has at its first position a supercomputer capable of performing slightly less than 150 *petaFLOPS* (i.e. $150 \cdot 10^{15}$ floating point operations (Flop) per second). Ten years before (November 2008), the Top500 list featured at its first position a supercomputer with only 1 *petaFLOPS* of performance, meaning an astounding growth of computing performance by more than 100 times over ten years. This growth is not only due to the improvements in the processor manufacturing process and transistor density, but it is as well due to the utilization of Graphics Processing Units (GPUs) [62] as processing accelerator devices – thus bringing heterogeneity as a standard in HPC supercomputers – and advancements in storage and interconnection technologies. Indeed, currently there is an active effort to achieve the *exaFLOPS* (10^{18} Flop/s) scale – the so called *exascale* – in the near future.

The increase in power consumption of supercomputers, however, has steadily increased as well, going from 2.4MW (2008) to 9.7MW (2018) [76] for the aforemen-

tioned top 1 supercomputers. To clarify the meaning of these numbers, 10 MW is arguably enough to power a small city.

From this perspective, we can argue that using supercomputers with the current technology is a matter of great responsibility: any inefficient usage of an HPC platform that consumes tens of megawatts of power can be considered a concerning matter. Increasing usage efficiency of current HPC platforms can be achieved in many ways, in which one of them is in the *resource management* aspect, that concerns in finding "good" answers for the question of when and where the applications will be processed in the platform.

This thesis goes towards advancing in this resource management aspect. Informally, a supercomputer infrastructure is usually a shared system: many *users* (research teams, companies, *etc.*) attempt to use the platform by running their *applications*, which are also mentioned in the literature as *jobs*. Jobs can be any computer program, though they are often programs that require a high computational power, such as the simulations and AI algorithms mentioned at the beginning of this chapter. Additionally, often many users try to use the platform in a concurrent manner – they submit their jobs for execution at the same time – and they can submit jobs at any unknown time. Such jobs also have processing times unknown in advance, further increasing the uncertainties.

As aforementioned, we can use the HPC platform more efficiently by performing optimal decisions as to when and where these jobs will be executed in the platform. This decision process is often referred as the *scheduling* of the jobs. Although it is easy to present informally, scheduling is a vast research topic (see Chapter 2), with many problems falling into the NP-Hard [64, 10] class of optimization problems, that hinds the conception of efficient algorithms, as well as the understanding of the problems' characteristics. In this regard, this thesis tackles a typical scheduling problem present in HPC platforms – the parallel job scheduling problem (see Section 2.1.2) – which is typically solved by many scheduler systems [61, 80] with algorithms inspired by a simple though naive First-Come-First-Served (FCFS) scheduling policy and a backfilling [60] mechanism, also referred to as EASY Backfilling (see Section 2.1.2.3).

Given the complexities and uncertainties involved in parallel job scheduling, there is a growing effort [40, 39, 52] to propose Machine Learning (ML) methods to improve scheduling performance.

1.1 Contributions

In this thesis we explore both using ML to perform a better resource management of HPC platforms, as well as better understanding the characteristics of some parallel job scheduling algorithms. More specifically, this thesis proposes the following broad sets of contributions:

1. We exploit the rich information present in HPC workload logs to drive simulations, whose objective is to generate data in regards to the scheduling behavior under many different scenarios. We then feed this generated data into Machine Learning models in order to learn potential scheduling heuristics – represented by nonlinear functions of the jobs' characteristics – present in the generated data;

2. We show that the learned heuristics can bring good scheduling performances in all evaluated scenarios, using real workload traces from highly different HPC platform configurations, when compared to classical and state-of-the-art *hand-engineered* scheduling policies;

3. We performed a careful experimental campaign to provide insight on possible expectations and performance gains if one replaces EASY Backfilling as the default on-line parallel job scheduling algorithm. Through simulations, we compared EASY Backfilling with a set of scheduling algorithms that have the same simplicity and job no-starvation guarantees as EASY Backfilling;

4. We conducted a holistic experimental analysis of the scheduling performances and show that, in many highly different HPC platform configurations, significant performance gains when compared to EASY Backfilling can be achieved, under many aspects and performance objectives, while maintaining the same simplicity and performance guarantees of EASY Backfilling.

1.2 Content

The remainder of this thesis is organized as follows: In Chapter 2 we provide background knowledge, notably about scheduling and machine learning, to introduce the reader to the contributions of the thesis, as well as we present the closely related works. Chapter 3 we present the aforementioned procedure of learning scheduling heuristics, with its respective experimental results and discussions. In Chapter 4 we present the aforementioned experimental campaign to provide insight on possible expectations and performance gains if one replaces EASY Backfilling, with its respective experimental results and discussions. At last but not least, in Chapter 5 we present a general discussion about the achieved contributions of this thesis, with the closing remarks and future research directions.

2 Background and Related Work

In this chapter we provide background knowledge to the reader to understand the work performed at the remaining chapters of this thesis, notably Chapters 3 and 4. This chapter contains background knowledge about scheduling problems and algorithms, as well as a brief introduction to machine learning, with emphasis on regression problems. Furthermore, to increase the presentation flow and quality of this chapter, many recent and related works are also presented in a intertwined manner along the text of this chapter.

2.1 Scheduling

Scheduling is a decision-making process that deals with the assignment of pieces of work (that need to be processed, generically referred to as *jobs*) to resources that will process the work at a given time period, and the goal of such decision-making process is to optimize an objective, in which can be constituted by one or multiple criteria.

Scheduling naturally relates to the organization of jobs and resources and such organization is clearly present in the majority of manufacturing systems, information processing, and even daily life environments, with jobs and resources taking different forms. For instance, the resources may be manufacturing machines, runways at an airport, processing units in a computer environment, crews at a construction site, and so on. In its turn, the jobs may be operations in a manufacturing process, take-offs and landings at an airport, computer programs to be executed, stages in a construction project, and so on.

The jobs may have several properties, different priorities, starting times and possible due dates. The objectives may also be different according to each scenario. One may want to minimize the completion time of the last job, and other may want to

minimize the number of delayed jobs, that is, the jobs finished after their respective due dates. We illustrate the importance of scheduling in the following, simple daily life example: supermarket checkout.

Supermarket Checkout: Consider a supermarket that sells a variety of items to customers. Customers select one or more items that they want to buy and, after they are done selecting their items, they arrive at the supermarket's checkout section with their shopping items to be paid. The checkout section can be constituted by one or more cashiers. The cashiers may service customers at different speeds (an experienced cashier may work faster than a beginner one), and can service only one customer at a given time. Once a cashier started servicing a customer, the checkout process can not be suspended, it must go until its completion. The customers can arrive in the checkout section uninterruptedly and at an unknown rate (when the supermarket is open) or all the customers to be serviced at the checkout section are known in advance (when the supermarket closes for new customers at the end of a working day, the customers will be only the ones already in the supermarket). The scheduling problem arises as the customer-cashier assignment in order to minimize some objective. The objective may be for instance the minimization of the average customers' waiting time at the checkout section (customers will overall wait as little time as possible), the maximum waiting time (no one will wait for too long) or even, when the supermarket closes, the objective may be to guarantee that the last customer will be checked out as quickly as possible.

One may observe that many distinct scheduling problems may be constructed considering a single scenario. Indeed, there is formal notation of scheduling problems that was initially conceived by Graham *et al.* [43], whose parts that are relevant to this thesis are presented in the next section. Besides Graham's original work, the reader can consult the works of Pinedo [64] or Brucker [10] for a full notation description.

2.1.1 Summary of Scheduling Framework and Notation

In most of scheduling problems the number of jobs and resources are assumed to be finite. The number of jobs is usually denoted by n and the number of resources (often referred to as machines) by m. The subscript j usually refers to a job, while the subscript i refers to a machine.

A scheduling problem can be described by a triplet $\alpha \mid \beta \mid \gamma$. The α field describes the machine environment, the β field details the processing characteristics and constraints, and the γ field describes the objective to be minimized.

The α field usually has only a single entry. Possible machine environments entries for α can be the following:

- *Single machine* (1). This case the simplest among all possible machine environments and it is a special case of all other more complex machine environments;
- *Identical machines in parallel* (P_m). This is the case where there is m machines in parallel that can be assigned to jobs. In this case, the processing speeds v_{ij} are the same for all machines i, $1 \leq i \leq m$ and jobs j, $1 \leq j \leq n$;
- *Parallel machines with different speeds* (Q_m). In this case there is also m machines in parallel that can be assigned to jobs. However, the processing speed depends on the machine i. Therefore, v_{ij} can be different for different machines i, $1 \leq i \leq m$, though for given a machine i, v_{ij} is equal for all jobs j, j, $1 \leq j \leq n$;
- *Unrelated machines in parallel* (R_m) In this case there is also m machines in parallel that can be assigned to jobs. However the processing speed depends on both the machines and jobs. Therefore, v_{ij} can be different for all machines i, $1 \leq i \leq m$ and jobs j, $1 \leq j \leq n$.

The β field refers to specific characteristics and restrictions of a scheduling problem. It may contain many entries and some possible entries are the following:

- *Release dates* (r_j). This symbol marks that a job j cannot start its processing time before its release date r_j. If this symbol is not present, then the job j can start its processing at any time;

- *Parallel jobs* ($size_j$). This symbol indicates that the jobs may require more than one machine at the same time (in parallel) in order to be processed. In this case, more than one machine may be simultaneously allocated for j;

- *Preemptions* ($prmp$). This symbol implies that a job j, once started its processing, does not need to be processed until its completion. In this setting, the processing of j can be interrupted (preempted) at any point in time, and another job can be assigned to the machine that was processing j. The amount of processing done is not lost when j is preempted. When j is put back on processing, it will only require the remaining processing time;

- *Precedence constraints* ($prec$). This symbol refers to the scenario that, in order to start the processing of a job j, one or more jobs must be previously executed as a requirement to start the execution of j.

The γ field refers to the objective function to be minimized in a scheduling problem and it can have one or more entries. The objective is often a function of the completion time of the jobs, denoted by C_j for a job j. For the sake of simplicity, the objective functions presented below are defined for scheduling problems where no preemptions are allowed.

Some of the classic objective functions are the *makespan* (C_{max}), which is defined as max($C_1, ..., C_n$), and it relates to the completion time of the last job that leaves the system, and the *sum of the completion time*, defined as $\sum_{j=1}^{n} C_j$, which relates to the completion time of all jobs. A minimum makespan indicates that the last job leaves the system as quickly as possible, whereas a minimum sum of the completion time indicates that all jobs leave the system as quickly as possible. In the literature, the makespan is well known by its analytical tractability. Many scheduling problems that aim to minimize the makespan have optimal algorithms [10] that is not just a brute force of all possible solutions.

The makespan is specially useful when the whole set of jobs and the number of jobs n to be scheduled are known in advance, this setting is referred in the literature as *static* or *offline*. While many practical problems fall in this offline category, the scheduling problem studied by this thesis (see Section 2.1.2) falls in the *dynamic* or *online* category, where the whole set of jobs and the number of jobs n are unknown in advance by the scheduler. In an online scenario, data about a job j is only known when the job is released. Data associated with a job j are often the following:

- The *processing time* p_{ij} is the time that job j takes to be processed at machine i. The subscript i is omitted if the processing time of j does not depend on the machine i. In theoretical cases p_{ij} is considered to be known in advance, whereas in most practical cases p_{ij} is either uncertain or unknown in advance;

- The *release date* r_j, also referred to as the ready date, is the time that job j arrives in the system. In other words, r_j is the earliest time at which job j can start its processing;

- The *number of machines* q_j required to process j is often associated in cases where a job j can use multiple machines at the same time in its processing. q_j is specially useful to characterize parallel jobs;

- The *processing time estimate* $\tilde{p}_{ij}$ is an estimation on the time that job j takes to process at machine i. The subscript i is omitted if the estimation does not depend on machine i. $\tilde{p}_{ij}$ if used to characterize jobs in cases where p_{ij} is not known in advance, and in some practical studies (see Section 2.1.2) $\tilde{p}_{ij}$ is also considered as an upper bound for p_{ij}.

It is not straightforward to set makespan as an objective function in an online scenario, as it is hard to define a last job in a system where the whole set of jobs to be schedule is unknown. In this case, the efficiency of an online scheduling algorithm is evaluated in theoretical studies by its *competitive ratio* in regards to an underlying objective function. An online scheduling algorithm is ρ-competitive if for any problem instance, the objective value of the resulting schedule created by the online algorithm is at most ρ times larger than the optimal objective value in the

case where the schedule had been created in an offline manner, with all data known beforehand [64].

Moving on more elaborated objective functions, many online scheduling problems aim to minimize objective functions related to the time that the jobs spent in the system. The basic component of these objective functions is the *waiting time* (Equation 2.1), where s_j is the time that a job j started its processing.

$$w_j = s_j - r_j \tag{2.1}$$

With only taking the waiting time into consideration, we can devise two objective functions, the average waiting time and maximum waiting time (Equations 2.2 and 2.3, respectively). The set of jobs J is often set as the set of jobs that finished processing at a certain time period. Typical time periods are in order of days, weeks or months. It is important to note that the set of jobs J is only a subset of the whole set of jobs to be scheduled, which remains unknown in an online scenario. This approach is an alternative to the competitive ratio to evaluate online scheduling algorithms.

$$\mathsf{AVGwait} = \frac{1}{|J|} \sum_{j \in J} w_j \tag{2.2}$$

$$\mathsf{MAXwait} = \max\left(w_1, ..., w_{|J|}\right) \tag{2.3}$$

Naturally, the waiting time itself is oblivious to the amount of processing time that the jobs require. Going back to the supermarket checkout example, while it is interesting to minimize the average time that the customers wait to be serviced by a cashier, one may argue that – assuming that the cashier service time is proportional to the number of items in a customer's shopping basket – customers with a shopping basket with few items should wait less than customers with a shopping basket with

plenty of items. Therefore, an equally important objective function would be one that accounts an important factor: the processing time.

In this regard, two objective criteria are often considered in online scheduling problems: the *flow time* (Equation 2.4, also referred as turnaround time), which derives the average flow time (Equation 2.5) and maximum flow time (Equation 2.6) objective functions, and the *slowdown* (Equation 2.7, also referred as stretch), which derives the average slowdown (Equation 2.8) and maximum slowdown (Equation 2.9) objective functions.

$$F_j = w_j + p_j \tag{2.4}$$

$$\mathsf{AVGF} = \frac{1}{|J|} \sum_{j \in J} F_j \tag{2.5}$$

$$\mathsf{MAXF} = \max\left(F_1, ..., F_{|J|}\right) \tag{2.6}$$

$$sld_j = \frac{w_j + p_j}{p_j} \tag{2.7}$$

$$\mathsf{AVGsld} = \frac{1}{|J|} \sum_{j \in J} sld_j \tag{2.8}$$

$$\mathsf{MAXsld} = \max\left(sld_1, ..., sld_{|J|}\right) \tag{2.9}$$

The flow time measures the total amount of time that a job spends on the system. In our supermarket checkout example, the flow time of a customer is the time that such customer waited at the checkout section to be serviced, plus the checkout service time. Minimizing the flow time means that the jobs will stay in the system as short as possible.

The slowdown, being the ratio between the flow time and the processing time, gives the idea that the waiting time of a job should be proportional to its processing time. A large job could "afford" to wait more and vice versa. For instance, a slowdown of 3 means that a job waited three times its processing time. However, it is noteworthy that the slowdown can be sensible for smaller jobs. Increasing the slowdown of a small job can be significantly easier than a larger job. Indeed, the slowdown can reach very high values for very small jobs. In this regard, many people opt to use the slowdown in its bounded version (Equation 3.1, see Section 3.2) by giving a lower bound on the processing time of the jobs (for instance, 10 seconds). Jobs with processing time lower than the bound will have their slowdown calculated taking into account the lower bound rather than their actual processing time.

As it can be observed, most of the objective functions have their average and maximum variants. Naturally, the average variants aim for a minimization of the objective function under an overall view in regards to the jobs (the average over many jobs), whereas the maximum variants aim at minimizing the objective function under a specific view in regards to the jobs (only a single job, the one that achieved the highest objective value). It is debatable which variant puts stricter constraints into the scheduling, although some argue that the maximum variants are stricter. A further discussion about the average and maximum variants of some of the aforementioned objective functions is presented at Section 4.3.3.

Many other objective functions can be envisioned, which can be centered or not at the jobs. One may conceive objective functions that are centered at the platform, such as overall utilization of the resources, or centered at the users, such as overall user "satisfaction". However, drawbacks will always exist for both approaches. For the former, a low overall resource utilization may not be due to the scheduler's performance and rather due to the amount of workload available [34]. For the latter, there may be user-user and user-platform feedback effects [26] that may have a bigger impact than the scheduler performance, and such feedback effects are hard to understand and/or reproduce. A further discussion on these feedback effects is done in Section 4.1.1.

One interesting aspect of the scheduling problems is that changing and/or adding entries into any of the α, β or γ fields may drastically change the problem's complexity, going from a problem with known polynomial-time algorithms to solve it, to a NP-Hard [38] problem, which polynomial-time algorithms to solve it are not yet known.

Indeed, there is a considerable amount of effort to establish a complexity hierarchy of scheduling problems, which attempts to determine an order of the α, β and γ entries that make the resulting scheduling problem easier or harder to be solved. Further dissertating about this hierarchy and other scheduling problems is beyond the scope of this thesis. The reader can consult Pinedo [64] or Brucker [10] for a more comprehensive explanation in this regard.

In the next section we are going to present with more detail a specific class scheduling problems, the online parallel job scheduling problems.

2.1.2 Online Parallel Job Scheduling Problems

With the increasing development of parallel algorithms and architectures, it is quite common that HPC applications may require multiple processing units in order to be processed. As a result of this and many other factors, today most of the HPC platforms are constituted by large-scale parallel machines, often with heterogeneous processing units. Additionally, often these parallel machines are a shared system, in the sense that multiple applications (the jobs) compete for an exclusive usage of a partition of the machine, and these applications are submitted at any point in time – and often unknown in advance – by many HPC platform users.

At this light we can devise some of the characteristics of online parallel job scheduling problems:

- As its name already suggests, online parallel job scheduling problems involve parallel jobs and machines. In regards to the former, there exist many modern HPC applications that use parallelism, specially after the advent of parallelization standards such as MPI [33] and OpenMP [17]. Therefore, parallel jobs

($size_j$ entry in the β field) are always taken into account. In regards to the latter, many machine configurations can be considered (notably the α entry of the problem). For a homogeneous machine, which is the case for machines constituted by a set of interconnected identical CPUs, it is considered the P_m entry. For heterogeneous machines, which are machines constituted by non-identical CPUs, the Q_m entry is considered. At last, with the advent of accelerator devices such as GPUs [62], MICs [20] and FPGAs [23] – jobs may choose to be processed on either a CPU or an accelerator device (if the respective binary executable is available, which may also be different for each accelerator device). This therefore raises the possible consideration of unrelated machines (R_m), since the speed of processing the job depends on both processing unit and job;

- In the context of modern HPC platforms, parallel job scheduling problems are inherently online, since it is often not known when the jobs will be submitted and, since the jobs can arrive at any point in time, the r_j entry of the β field is also considered;

- Job preemptions (the $prmp$ entry in the β field) are often not allowed, since yet many HPC applications does not support preemption. However, preemption may be allowed in a Cloud Computing or Big Data scenario, where preemption support is frequently seen as a requirement for their applications;

- Typically jobs are seen as independent from each other, and hence with no precedence constraints ($prec$ entry in the β field). However, in the case where applications are constituted by job workflows [81, 16], precedence constraints may be accounted;

- The objective function of the γ field can be either metrics centered on the jobs, such as waiting time, flow time and slowdown (see Section 2.1.1) or metrics centered on the users, such as fair share of resources or user "satisfaction" (see Section 4.1.1).

There is an additional characteristic in regards to the jobs, in which can be *rigid* or *moldable* [30, 14]: a job j is considered to be rigid if no less than q_j resources must be reserved for j in other to be processed. In the converse case, where j can be processed with less than q_j resources, j is considered moldable. Jobs are often considered to be rigid in a standard HPC scenario, though they can be considered moldable in cases such as in a cloud computing scenario, where job moldability is an important requirement.

A final schedule generated by a scheduling algorithm can be visualized by a graph called Gantt chart. Figure 2.1 illustrates an example, with a workload consisting of 250 jobs scheduled in a HPC platform with 128 processors. The processors are represented by their ids in the y axis (from 0 to 127) and the time is represented at the x axis. A job j being processed in the platform is represented by q_j rectangles of unit height (y axis), with each having p_j of width (x axis), and all starting in the x axis at s_j. The rectangles that represent a job j may or may not be adjacent to each other and, in the former case, the adjacent rectangles are visualized as "boxes" in the xy plane.

Gantt charts are a powerful tool to visualize schedules. For instance we can easily observe and have an idea about the overall resource utilization of the HPC platform by just looking on how many "empty spaces" exist in a Gantt chart. However, it only represents a static view of the final schedule, other factors such as number of jobs in the waiting queue and workload submission rate (i.e. number of job arrivals at a certain time period) are not well represented by a Gantt chart.

Gantt charts also highlight some assumptions that are made in parallel job scheduling problems. For instance factors such as the jobs' level of parallelism, interconnection network topology and possible interconnection network contentions caused by the jobs, which can impact the jobs' processing time [6] are sometimes not taken into account in the scheduling decisions. The jobs are rather seen as "black boxes" that exclusively use q_j processors during p_j units of time. Although this context obliviousness in regards to the jobs is the default assumption in many scheduling

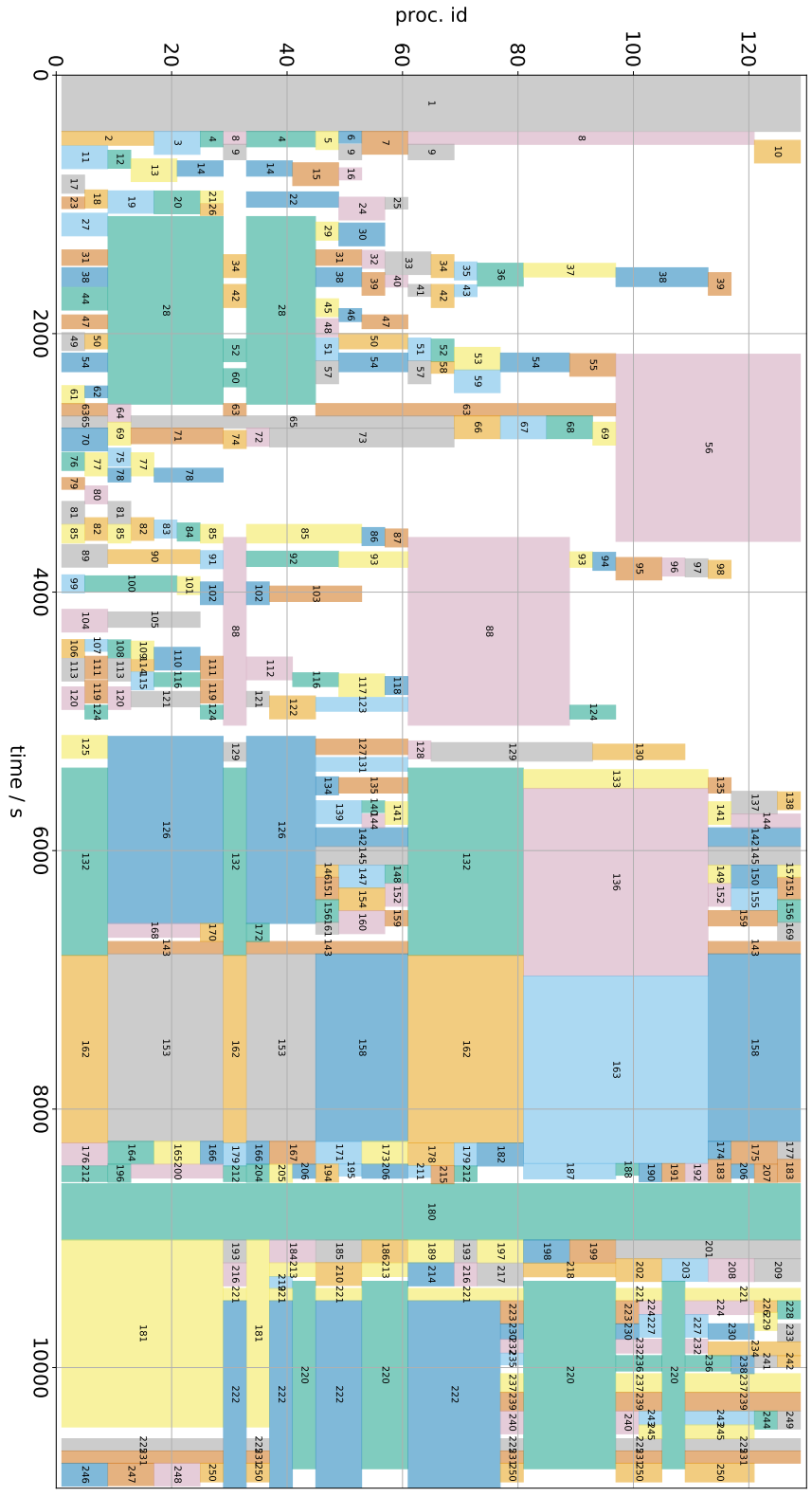


Figure 2.1 Gantt chart of a typical HPC workload. Jobs executed by processors (y-axis) are depicted by rectangles along the time line (x-axis). This chart represents the execution of 250 jobs on a platform with 128 processors. Figure obtained from [8] with permission from the original author.

studies (including the one in this thesis) there are some initiatives that try to bring context awareness to scheduling [8, 37].

Parallel job scheduling problems are largely studied under the view of a more general and closely related problem called multiple-strip packing problem [2] in which, informally, aims to find a packing configuration of rectangles (jobs) into a set of strips (set of processors) in order to minimize the maximum height of the rectangles' packing, and rectangles rotations are not allowed. It is known that the single strip case is NP-hard [2] and, thus, many approximation algorithms – where the performance is worse than the optimal, though guaranteed to be within a certain distance from the optimal – and performance bounds were proposed [9, 79, 44, 82, 77].

A noticeable amount of research is employed into offline parallel job scheduling problems with objectives such as makespan and total completion time, all of them as well focused on approximation algorithms [1, 46, 45], since even simple offline parallel job scheduling problems such as $P_m \mid size_j \mid C_{max}$ and $P_2 \mid r_j, size_j \mid C_{max}$ are NP-Hard [50, 19].

Into the online, non-preemptive scenario, and with complex objective functions such as the ones involving slowdown, conceiving theoretically efficient scheduling algorithms is being a challenging endeavor, mainly due to the strong theoretical performance bounds [48, 3] found even for simple, single machine problems. Nevertheless, fortunately there are still theoretical initiatives [51, 22, 57] that work in these parallel job scheduling problems.

In the practical scenario, the majority of Resources and Jobs Management Systems (RJMS, see Section 2.1.2.2) employ list scheduling based algorithms, which are discussed in the next section.

2.1.2.1 List Scheduling Algorithms

List scheduling algorithms rely on a very simple idea, which consists on evaluating all jobs in a waiting queue according to a criterion, and the job that is selected

to be processed is the job that achieved the best value according to the criterion used. More specifically, a list scheduling algorithm sorts – in increasing or decreasing order according to a scheduling policy $f(j)$ – the jobs in its waiting queue, and this sorting is performed when scheduling decisions are needed, which are typically in two distinct events: (i) when a job arrives in the queue or (ii) when a resource (set of processors) is released and becomes available. A job j receives the highest priority to be processed if j is the first in the sorted waiting queue. When a job j is selected to be processed and if the requested number of processors q_j is lower than the total number of processors available, then q_j processors are reserved for this job and they become unavailable. These processors will become available again only when p_j units of time have passed since the start of the processing of j. In some list scheduling configurations (specially in Backfilling scheduling, see Section 2.1.2.3), if the actual processing time p_j is larger than its estimate $\tilde{p}_j$, j is *killed*, that is, its execution is terminated and all the processing performed for j is lost.

If q_j is higher than the number of processors available, then the scheduler waits either events (i) or (ii) to perform another scheduling decision, since it cannot immediately dispatch j for processing. Instead of waiting, however, many list based schedulers perform a *backfilling* mechanism, in an attempt to use the idle available processors without delaying the processing of the jobs in the queue. We further elaborate this point at the end of this section.

A key component of a list scheduling algorithm is the scheduling policy $f(j)$, which is the component that encapsulates the envisioned scheduling criterion. Usually a scheduling policy $f(j)$ is a function with the characteristics of a job j as input, and a numeric value as output, which represents the priority of j. Following this definition, there is naturally a very large number of possible scheduling policies.

Arguably the most intuitive policies are the ones that take only one characteristic of the jobs into account, notably $f(j) = r_j$, $f(j) = \tilde{p}_j$, and $f(j) = q_j$. A waiting queue sorted in increasing order by these functions yields the First-Come-First-Served (FCFS), Shortest Estimated Processing Time First (SPF) and Shortest Processing Requirement First (SQF) scheduling policies, respectively. Conversely, a waiting

queue sorted in decreasing order by these functions yields the Last-Come-First-Served (LCFS), Largest Estimated Processing Time First (LPF) and Largest Processing Requirement First (LQF) scheduling policies, respectively. Although their scheduling performances vary (see Chapters 3 and 4), FCFS is by far the most chosen policy by practitioners and, specifically for parallel job scheduling problems, the "last/largest" variants consistently perform worse than the "first/shortest" variants [39].

Other quite intuitive policies are the ones that take the two main dimensions of parallel jobs ($\tilde{p}_j$, and q_j) into account in a straightforward manner, notably $f(j) = \tilde{p}_j \cdot q_j$, which yields the Shortest Estimated Area First (SAF) and the Largest Estimated Area First (LAF) scheduling policies, for a waiting queue sorted in increasing or decreasing order, respectively. These policies naturally relate to how parallel jobs are represented in a Gantt chart. Jobs are sorted according to the "geometry" or "area" of their rectangles in a Gantt chart. In Chapter 4 we investigate one of these policies (SAF) and we show experimental evidence that this policy is an efficient scheduling policy.

Going on more complex scheduling policies, there also exist hand-engineered policies such as the ones proposed by Tang *et al.* [75] that encapsulate some complex scheduling intuitions and, expectedly, the more complex these policies are, the less simple and transparent these policies become. In Chapter 3 we seek ways to automatically *learn* scheduling policies using data regarding to the platform and workload characteristics, while maintaining a certain level of simplicity and transparency.

In the above description, list scheduling algorithms are described in a simplistic view. As mentioned above, modern scheduler systems (see Section 2.1.2.2) employ a variant of the previously mentioned list scheduling algorithm, whose difference stands in the case where a job j is selected to be processed, though there are not enough available processors to process j. In this case, an *aggressive or conservative backfilling* subroutine [60] is often applied. Informally, a backfilling subroutine estimates at which time there will be enough resources to process j. Next, the scheduler looks for jobs j_b in the waiting queue – following the order of jobs already

established by the scheduling policy $f(j)$ – for which there are enough processing resources and that do not delay the execution of either only j (the aggressive case) or all jobs with higher priority than j_b (the conservative case). If j_b meets these aforementioned conditions, then j_b "jumps ahead" and is scheduled for processing. The general idea of backfilling techniques is to "fill the empty spaces" in a schedule visualized by a Gantt chart, thus increasing the overall resource utilization of the HPC platform, while also maintaining the reasoning behind the underlying scheduling policy. Indeed, backfilling algorithms have shown to be effective at this task, to the point that many recent works in list scheduling algorithms (including the ones in this thesis) employ backfilling in some way. More details of backfilling algorithms and some of the aforementioned recent works are presented in Section 2.1.2.3.

Once a job j receives the highest priority and it is scheduled for processing, the resource allocation for j can be performed in many ways. Arguably the simplest approach is to allocate any partition of q_j available processors, thus not assuming any contiguity constraints. However, choosing any partition may not be appropriate for all scenarios, specially in scenarios where communication and contiguity affects the jobs' performance. In this thesis we are focused on the job prioritization part of list scheduling algorithms, assuming a resource allocation policy with no resource contiguity constraints. The reader can consult Lucarelli *et al.* [56] for a comprehensive study on resource allocation contiguity.

2.1.2.2 Resources and Jobs Management Systems

It is a common practice that HPC platform administrators deploy into their platforms a system software called Resources and Jobs Management System (RJMS). As its name suggests, the RJMS is responsible of managing and brokering all interaction between the HPC platform users and the HPC platform itself. It not only manages which and when the jobs will be processed (job prioritization), but as well which resources the jobs will use (resource allocation).

Figure 2.2 illustrates a typical RJMS configuration. Users submit their jobs through the RJMS jobs manager module, which is responsible to manage the submitted jobs.

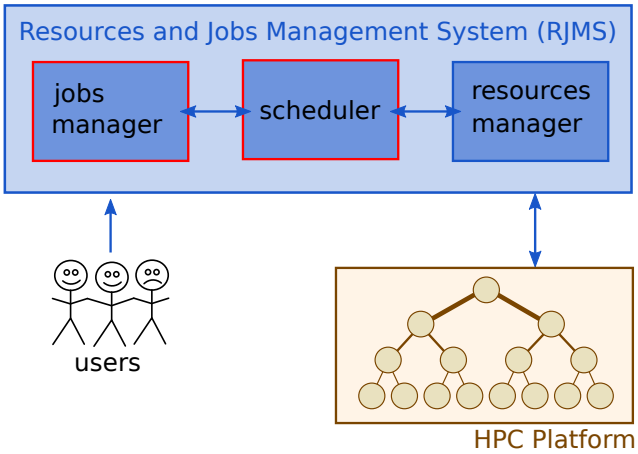


Figure 2.2 Typical Resources and Jobs Management System Configuration. Figure adapted from [65] with permission from the original author. The modules highlighted in red are the ones studied in this thesis.

At the job submission, the RJMS has limited information about the jobs. Common information available are the number of processors required by the job, the estimated processing time $\tilde{p}_j$ and the command script to the executed by the job. The jobs manager module also provides jobs' information to the scheduler module, whose responsibility is to decide when the available jobs will be executed.

The scheduler's task mainly involves performing a prioritization of the submitted jobs, and it is the module that actually solves the parallel job scheduling problem. In this regard, a noticeable phenomenon is the divergence between theory and practice [30]. With the introduction of backfilling algorithms [60] – whose reasoning is to allow a job with low priority to be executed before a higher priority job if this low priority task does not delay the higher priority one – in practice most scheduler modules use algorithms inspired by either FCFS with aggressive backfilling (called EASY Backfilling [60] algorithm) or the same aggressive backfilling with some scheduling policy to sort the waiting jobs. A detailed description of backfilling algorithms is presented in Section 2.1.2.3.

For instance, the SLURM job management system, a well known RJMS for HPC platforms [80], uses a multi-factor sorting of the waiting queue, using a linear combination of priority factors (jobs waiting time, tasks size, share factors, *etc.*), with coefficients – whose values establish the relative importance of these priority factors – defined by the HPC platform maintainer, and then performs backfilling over

the sorted queue. Other RJMSs such as TORQUE [72], PBSpro [61], MOAB [15] or OAR [11] implement similar approaches with their own specificities. Some HPC platforms [66] simply choose to set a multiple job queue paradigm, where each queue has different priorities in which are arbitrarily set by the HPC platform maintainer, and the jobs whithin a queue follow FCFS order.

Many reasons can be devised to justify the choice of simple heuristics, notably EASY Backfilling, as the scheduling algorithm: it is established that EASY Backfilling increases the overall utilization of the platform, while keeping a relative simplicity and job starvation guarantees [60]. Furthermore, although it is also established that there is room for improvement in the scheduling, replacing EASY Backfilling with another algorithm might be seen as a risky change: one can see this change as a "jump into the dark", with the changes in performance only noticeable after a long period of time, and potentially after many strong-worded emails from many (important) users.

This unwillingness to apply different policies other than EASY Backfilling is arguably due to the lack of clarity and interpretability of these policies, making the whole scheduling algorithm less transparent to the users. In Chapter 4 we aim to bring light to this subject, by arguing and showing experimental evidence that some class of equally simple scheduling algorithms can provide significantly better performances in many aspects of the scheduling, when compared to EASY Backfilling.

At last, the resources management module receives information from the scheduler about the selected jobs and manages the resource allocation for such job. The resources management module also keeps track on the available resources, notably the availability of the computing nodes. In this thesis we are more focused on the scheduler part of a RJMS. The reader can consult Georgiou [41] for a detailed description of RJMSs and their characteristics.

2.1.2.3 Backfilling Algorithms

In a parallel job scheduling scheme, jobs receive a partition of the available resources for exclusive usage. A list scheduling algorithm (notably with a FCFS scheduling policy) may lead to resource under-utilization, specially in the case where the number of available resources is not enough to process the job with the highest priority. By default, these resources will remain idle until more resources are freed, eventually meeting the resource requirements of the highest priority job.

Prior research to solve this fragmentation problem were based on dynamic partitioning [59] and gang scheduling [28] strategies, though these strategies presented some practical limitations that hindered their deployment in practical scenarios. Solving this problem in practical scenarios was a challenging endeavor, and many practitioners ended up deploying only FCFS in their platforms, thus still having resource fragmentation issues. This was the default case until the advent of backfilling algorithms. Backfilling algorithms are based on the assumption that, in the case that there are not enough processors to process the highest priority job in the waiting queue, there may be lower priority jobs in the waiting queue that could "jump ahead" for processing, while also not delaying the processing of higher priority jobs and, by "filling the empty spaces" in the schedule, the resource fragmentation is lowered.

Backfilling algorithms were initially conceived following a simple FCFS list scheduling algorithm (see Section 2.1.2.1). In a non-backfilling list scheduling algorithm, if there are not enough resources to process the highest priority job in a waiting queue, the scheduler and the unused resources would remain idle until more resources are released, eventually satisfying the resource requirements of the highest priority job. Backfilling algorithms try to search for jobs to backfill at this idle period, and this search can be performed in two different ways which we informally present below. The reader can consult Mu'alem and Feitelson [60] for backfilling implementation details:

- *Conservative backfilling:* In this setting, a lower priority job j_b is backfilled from the waiting queue if its processing does not delay any job j with higher priority

than j_b. This setting is called conservative because it searches for jobs that does not harm at all the initial FCFS schedule for the higher priority jobs. In this case, the backfilled jobs will increase the resource utilization while maintaining all properties of the FCFS policy: all waiting jobs will eventually be executed (no starvation) and their waiting times are bounded [60]. However, conservative backfilling typically involves "reconstructing" the Gantt chart of the schedule considering each backfilling job j_b candidate and seeing if any higher priority job gets delayed or not. This whole process can be computationally expensive depending on the waiting queue and HPC platform sizes;

- *Aggressive backfilling:* In this setting, a lower priority job j_b is backfilled from the waiting queue if its processing does not delay only the highest priority job in the waiting queue, thus disregarding any higher priority job than j_b that is not the highest. The advantage of aggressive backfilling is that it is computationally cheaper than its conservative counterpart, though it does not provide the same properties of FCFS policy: it continues to prevent starvation, meaning that all jobs in the waiting queue will eventually be executed, however their waiting times are unbounded. For a job j, there may be an unbounded number of backfilled jobs, that can potentially delay the processing of j until j gets the highest priority [60]. One of the first deployments of aggressive backfilling with FCFS was in Argonne National Laboratory (ANL) to schedule their IBM SP1 parallel machine. They baptized their scheduling system as the Extensible Argonne Scheduling sYstem (EASY) [54] and, since then, FCFS with aggressive backfilling is often referred to as EASY-backfilling or just simply EASY.

Knowing whether jobs may or may not be backfilled – which involves reconstructing the schedule ahead of time as aforementioned – strongly relies on the jobs processing time estimates. For instance, an estimate lower than the actual processing time may mistakenly lead to backfilled jobs that will actually delay other jobs, including the one with the highest priority. Having accurate estimations of the processing time of the jobs has become such an important factor for backfilling schedulers to an extent

that these schedulers started to kill jobs if their actual execution time exceeds its estimate. This has led to a phenomenon that users started to always provide over estimations of the processing time of their jobs, thus making the processing time estimate a good upper bound for the actual processing time, though nevertheless the accuracy of the estimations remained to be poor [27].

More recently, a considerable amount of research was devoted to improve the performance of backfilling algorithms by adopting a two-queue paradigm (one primary queue and a secondary queue to search for backfilling jobs) and performing a tuning of these queues [53, 63, 73]. More recently, Gaussier *et al.* [40] explored the usage of machine learning in backfilling and proposed a variant of the EASY algorithm that relies in a job processing time prediction model to perform the backfilling decisions rather than using the estimates provided by the users. In this thesis we consider the aggressive backfilling as the baseline backfilling mechanism.

2.2 Machine Learning

It is perhaps intrinsic of the human nature to try finding patterns in a set of observations of a certain phenomenon. Possible motivations are just to understand how the phenomenon works or to perform predictions about future events around the phenomenon. Whichever the motivations are, the interest in finding such patterns was arguably a key component to develop the science we know today.

Throughout human history, there are many records on successful attempts in obtaining knowledge from patterns found in observations, such as the ones obtained by Kepler [49], which has resulted in the development of modern astronomy, and Darwin [18], which has resulted in the modern biology we know today. In a less sounding manner, there are as many records of unsuccessful attempts to find patterns in observations, such as the prior work performed by Tycho Brahe on how celestial bodies move in the space, whose observational data helped Kepler to devise simple laws on how celestial bodies move.

All of these records indicate how difficult is to find these patterns and, therefore, how difficult is to obtain knowledge from observations. In Kepler's case, it was not only necessary to advocate on pioneer insights at his time (the Earth moves around the Sun and not the opposite), but it was necessary as well to solve rather complex and cumbersome calculations in order to devise three laws, the three Kepler's Laws. With the advent of modern computers, a lot of progress has been done to conceive automatic ways to find patterns in data, thus giving birth of the machine learning (ML) research field that we know today.

With automatic ways of finding patterns into data, it has risen the impression that every problem or phenomenon can be solved or understood if we have more data about it. This has resulted in a development loop on ways to obtain more data, and more efficient ways to automatically find patterns into this data. It boils down that pioneer insights, such as the ones Kepler had based, are not completely (perhaps not at all) replaced by automatic procedures, though such automatic ways to find patterns in data has helped on progresses that would not be feasible by hand.

Problems tackled by machine learning are typically classified into two broad class of problems: *supervised* and *unsupervised*. In supervised problems, data consisting of a set of observed examples, each one having its inputs (often referred to as *features*) and its respective *desired output* are available beforehand, so that prediction models can be trained based on these data. In regards to the desired output, it can be a category among a finite set of categories. In this case the machine learning problem is referred to as a *classification* problem. However, the desired output can also be one or more continuous values. In this case, such machine learning problem is referred to as a *regression* problem.

In unsupervised problems the desired output is not known in advance, only the features of the examples are available. The goal in such problems may be to find groupings of similar examples into the data. Such problem is known as a *clustering* problem. Another unsupervised problem is to determine the distribution of data within the feature space, known as a *density estimation* problem.

For both supervised and unsupervised problems, many machine learning algorithms and models have been proposed and its number grows steadily. More complex models and also more algorithms to fit these models into the observed data are proposed in the literature almost daily. However, having a machine learning model that is transparent – in the sense that the semantics behind its predictions are clear – and also powerful in prediction performance is still a challenging endeavor for some problems.

As a result, deciding which model to use does not only depends on the technical constraints of the models and algorithms but as well depends on the *purpose* of the learned model. If the purpose is to get a better understanding of the phenomenon under study, and thus prediction performance is not as important, using a less powerful and more transparent model may be more appropriate. However, if the purpose is prediction performance, where understanding the reasons behind the model's decisions may not be as important, using a more powerful and less transparent model may be more appropriate.

In the next section we introduce simple models and algorithms for linear and non-linear regression problems, which was the problem modeling used in Chapter 3. The reader can consult Bishop [7], Friedman *et al.* [35] or Berthold *et al.* [5] for a detailed presentation of classic machine learning models and algorithms, and Goodfellow *et al.* [42] for more advanced, state-of-the-art machine learning models and algorithms.

2.2.1 Regression Problems

As mentioned in the previous section, the goal in regression problems is to perform predictions of one or more continuous variables given the observed input features. More specifically, given an observation consisting of an input vector $\mathbf{x}$ of features, the objective is to predict one or more continuous *target* variables t given the values present in $\mathbf{x}$. This problem setting can be used to model many practical problems, for instance predicting crop yield given a set of data such as sunlight intensity and pluvial frequency, and – more specifically related to Chapter 3 – predicting how

much beneficial would be for the scheduling performance if a certain job from the waiting queue would be selected for processing.

Regression problems are situated in the supervised learning class of problems. This means that a *training* data set constituted by N observations $\{\mathbf{x}_n\}$, where $n = 1, \ldots, N$, and their corresponding target values $\{t_n\}$ are known in advance, and the objective is to perform predictions of t for new, unseen inputs $\mathbf{x}$, taking into account the previous patterns present in the training data.

An initial approach to solve this prediction problem is to construct a prediction model consisted by a function $y(\mathbf{x})$, whose output for a new observation $\mathbf{x}$ constitute the target predictions t for $\mathbf{x}$. This typically involves designing two main components: (i) the type of function $y(\mathbf{x})$ to be used, and (ii) an appropriate optimization algorithm, to adequately fit $y(\mathbf{x})$ to the patterns present in the training data. A key hypothesis in this approach is that the patterns present in the training data are relevant to perform predictions to new, unseen observations.

In order to bring flexibility for $y(\mathbf{x})$ to fit into the training data and, more importantly, to reduce the search space to avoid overfitting (see more at the end of this section) in cases that there is not enough training data to perform a good fit, the function $y(\mathbf{x})$ is actually defined in a parametric form $y(\mathbf{x}, \theta)$, where θ is a vector of parameters.

Given an input vector $\mathbf{x}$ with D input features $x_i, 1 \leq i \leq D$, arguably one of the simplest forms of functions $y(\mathbf{x}, \theta)$ are functions of the following form,

$$y(\mathbf{x}, \theta) = \theta_0 + \theta_1 x_1 + \ldots + \theta_D x_D \tag{2.10}$$

where θ_0 is an additional parameter to add a fixed offset flexibility for $y(\mathbf{x}, \theta)$, and it is often called as *bias* or *intercept* parameter. It is important to observe that Equation 2.10 is linear on both the coefficients in θ and the features in $\mathbf{x}$. One alternative to bring more nonlinear flexibility is to maintain $y(\mathbf{x}, \theta)$ linear on the

coefficients in θ, though nonlinear in respect to the features in $\mathbf{x}$. This gives the following function,

$$y(\mathbf{x}, \theta) = \theta_0 + \sum_{j=1}^{M-1} \theta_j \phi_j(\mathbf{x}) \tag{2.11}$$

where $\phi_j(\mathbf{x})$ is a nonlinear function of $\mathbf{x}$. M is a natural positive number that can be set arbitrarily, and controls the number of parameters – as opposed to Equation 2.10, in which the number of parameters is controlled by the number of input features D – and the number of nonlinear transformations present in the model. Equations 2.10 and 2.11 are considered cases of *linear regression*, even though Equation 2.11 can represent non linear relationships in function of $\mathbf{x}$. The concept of "linear" comes rather from the linearity in function of θ than in function of $\mathbf{x}$.

One interesting result in regards to Equations 2.10 and 2.11 is that considering a *training data set* constituted by N observations $\{\mathbf{x}_n\}$, where $n = 1, \ldots, N$, and their corresponding target values $\{t_n\}$, optimal parameters $\hat{\theta}$ can be found when optimizing for the sum of squared loss (Equations 2.12 and 2.13) in a process often called as least squares fitting [7, 5].

$$\Sigma_L = \sum_{n=1}^{N} L(y(\mathbf{x_n}, \theta), t_n) \tag{2.12}$$

$$L(y(\mathbf{x}, \theta), t) = (y(\mathbf{x}, \theta) - t)^2 \tag{2.13}$$

Although Equation 2.11 allows some non-linearity in function to $\mathbf{x}$, the linear constraint in function of θ can be a limiting factor to model some problems. In some scenarios, there may be prior insights saying that the relationship between the predictions and the input follows a particular functional form, rather than a linear combination of the coefficients. In this case, a function $y(\mathbf{x}, \theta)$ that depends nonlinearly on one or more coefficients in θ may be better suited. For instance, one may devise that a good list scheduling policy (see Section 2.1.2.1) can be a

function of the form $f(j) = \tilde{p_j}^{\theta_1} q_j^{\theta_2}$ for a job j. Performing regression in functions such as $f(j)$ results in a regression problem called as *nonlinear regression* [68]. With nonlinear regression, the same least squares fitting algorithms can be used, though the optimality of the coefficients θ in regards to the sum of the squared loss is no longer guaranteed.

There are more complex models for regression such as kernel methods [7] and neural networks [42]. Specifically for neural networks, with the current advancements specially in GPU computing [62], it is being possible to fit models with thousands of parameters, constituting noticeably powerful models. Although this model power may be required for some real-world problems, having a powerful model may lead to undesired phenomena such as *overfitting*. Informally, an overfitted model "memorizes" every single input and the desired target output present in the training data, and not actually being capable to provide accurate predictions to new, unseen data. Overfitting rises specially when one tries to fit a complex model (i.e. with many parameters) using a training data with low dimensionality (i.e. inputs with few features), or a training data with few examples.

Furthermore, as mentioned in the previous section, having a quite complex model may be detrimental to its transparency, as it is arguably hard to reason about how predictions are being made taking into account thousands of parameters. In this regard, linear and nonlinear regression models with few parameters are more advantageous with regards to transparency, as it is easier to grasp the contributions of each of the fitted parameters in the regression.

Yet, more complex models (notably neural networks) are being largely used and they are being successful in performing predictions in many complex real-world problems, specially in computer vision and natural language processing. In these cases, the models are rather seen as "black boxes", where only the prediction performance is taken into account. However, there is a growing interest in finding frameworks, such as the SHAP [58] framework, for interpreting and understanding how predictions are made with these complex models.

Once a model f is fit to the training data and their learned coefficients $\hat{\theta}$ are found, there are many metrics in regression that can evaluate the prediction performance of the f parametrized by $\hat{\theta}$. Considering a *testing data set* constituted by M observations $\{\mathbf{x}_m\}$, where $m = 1, \ldots, M$, and their corresponding target values $\{t_m\}$, which are unseen data for f during training, one of the simplest forms of evaluation is the Mean Absolute Error (MAE), which can be defined as follows:

$$\mathsf{MAE}(f) = \frac{1}{M} \sum_{m=1}^{M} \|f(\mathbf{x_m}, \hat{\theta}) - t_m\| \tag{2.14}$$

MAE closely relates to the sum of the squared loss (Equation 2.13) function, however, since there is not a squared penalization for the *residual* $f(\mathbf{x_m}, \hat{\theta}) - t_m$ (as opposed to the sum of squared loss), there is an equal emphasis on small or large errors, characterized by small or large residuals, respectively.

In regards to this squared penalization, another metric used in evaluating models is the Mean Squared Error (MSE), which can be defined as follows:

$$\mathsf{MSE}(f) = \frac{1}{M} \sum_{m=1}^{M} (f(\mathbf{x_m}, \hat{\theta}) - t_m)^2 \tag{2.15}$$

MSE is quite similar to the sum of squared loss, with the difference that MSE divides the sum of squared loss (residual) by the number of the testing examples M. As opposed to MAE, the Mean Squared Error emphasizes large residual values, given the squared factor applied to the residual.

For both MAE and MSE, the meaning of their values is straightforward: a value of 0.0 means perfect prediction performance and, the larger the value, the worse is the prediction performance of f. There is another prediction performance metric, however, that relativizes the prediction performance in function of the variance of

the target values present in $\{t_m\}$. Such metric is called Coefficient of Determination (or R^2), which can be defined as follows:

$$R^2(f) = 1 - \frac{\sum_{m=1}^{M}(f(\mathbf{x_m}, \hat{\theta}) - t_m)^2}{\sum_{m=1}^{M}(t_m - \bar{t})^2}, \tag{2.16}$$

where $\bar{t}$ is the mean of the target values present in $\{t_m\}$. A R^2 value of 1.0 means perfect prediction, any value in the interval $]0, 1[$ means not perfect prediction performance, with better prediction performance with values closer to 1.0. A R^2 of 0.0 means that f always predicts the target values, regardless of the features' values and, finally, any value lower than 0.0 means poor prediction performance.

2.3 Summary

In this chapter we presented a brief overview of the scheduling theory in the context of High Performance Computing, with emphasis on parallel job scheduling problems, list scheduling and backfilling algorithms. We also made a short introduction to machine learning, with emphasis on regression problems. Finding optimal solutions for parallel job scheduling problems is notoriously hard and, although there are many approximation algorithms with proven performance guarantees, many practitioners end up adopting simple heuristics, such as EASY Backfilling, to schedule parallel jobs. The content present in this chapter must be sufficient for the reader to follow the remaining chapters of this thesis. In the remaining chapters we present the main contributions of the thesis, starting by presenting the ways we explored simulation and machine learning to learn simple parallel job scheduling heuristics.

3 Machine Learning Discovery of Scheduling Policies[1]

3.1 Introduction

The on-line scheduling of jobs in large-scale HPC platforms is a complicate subject to be tackled by scheduling systems. The need to address numerous scheduling factors leads to the development of sophisticated scheduling algorithms that are often difficult to reason about or hard to deploy in real systems. As further elaborated in Section 2.1.2.1, an appealing alternative is to adopt scheduling policies – which are functions that take as input the characteristics of the jobs (*e.g.* processing time, requested number of processors, waiting time, *etc.*) and output a value that denotes the priority of the job – to perform the scheduling. These scheduling policies are often hand-engineered, generally based on intuitions (that may keep a clear understanding of what happens in the scheduling for the users) regarding which rules the scheduling policies must impose to achieve good scheduling performance. One common practice of the scheduler systems is to add a backfilling mechanism (see Section 2.1.2.3) in conjunction to the scheduling policy. The backfilling increases the utilization of the HPC platform and consistently improves scheduling performance.

Another common practice of HPC platform maintainers is to register information about the jobs that have been executed in the platform. These workload logs contain information regarding the characteristics of jobs, such as the ones mentioned above and processing time estimates provided by users. In light of the ever increasing amount of information generated by HPC platforms and the need for simple and efficient scheduling solutions, the main question raised in this chapter is: *Is it possible*

[1]The text of this chapter is adapted from the following published paper: **Danilo Carastan-Santos**, and Raphael Y. de Camargo. "*Obtaining dynamic scheduling policies with simulation and machine learning*". In the International Conference for High Performance Computing, Networking, Storage and Analysis (SC), **2017**.

to design an arguably simple procedure to learn general and simple scheduling policies from existing workload logs, which perform better than the existing hand-engineered scheduling policies?

In this chapter, we present a technique based on simulation and machine learning algorithms to generate simple scheduling policies represented by nonlinear functions. These nonlinear functions capture the characteristics of jobs that should be prioritized under several distinct situations and, when used as scheduling policies, improve the global scheduling of jobs. More specifically, this chapter presents the following contributions:

1. We show that it is possible to generate efficient scheduling policies in the form of simple though not obvious nonlinear functions, obtained from general workload characteristics, that improve global scheduling, when compared to classical and state-of-the-art hand-engineered scheduling policies;

2. We propose a simple simulation procedure and a machine learning strategy, based on nonlinear regression, to observe the effects of scheduling decisions over jobs obtained from a workload model over distinct conditions, and to model these effects into nonlinear functions that can be used as on-line scheduling policies;

3. We show that these obtained scheduling policies perform well when scheduling jobs from the same workload model used to observe the scheduling effects, with significantly better performances than the best performing hand-engineered scheduling performance in the most realistic setting evaluated (*i.e.* in conjunction with a backfilling algorithm and considering the user estimates of job processing times to perform scheduling decisions);

4. We show that the scheduling policies obtained by the procedure of the item 2 can generalize well, bringing good scheduling performances in all evaluated real world scenarios, using real workload traces from highly different HPC platform configurations.

The remainder of this chapter is organized as follows: in Section 3.2 we present the proposed strategy to obtain on-line scheduling policies. We present the main results (*i.e.* the obtained scheduling policies and its scheduling performances) in Section 3.3, and the summary in Section 3.4.

3.2 Finding Scheduling Policies with Simulation and Machine Learning

We tackled the on-line scheduling problem of executing a set of concurrent parallel jobs – whose resource requirements are fixed and known in advance (also known as rigid jobs) – on a HPC platform. The general idea proposed by this chapter is to design a simulation scheme to observe the scheduling behavior over sets of jobs under several distinct conditions and to use machine learning techniques to model the effects of job characteristics on scheduling performance into nonlinear functions. These functions can then be used by production on-line schedulers to determine – based on jobs characteristics, such as estimated processing time, resource requirements and submit time – the next job to choose from the queue for execution.

3.2.1 Scheduling Background

In this Section we briefly summarize the background and assumptions of the parallel job scheduling problem under study in this chapter. The reader can consult Chapter 2 for a more detailed and broader description of scheduling problems.

We consider an HPC platform as constituted by a set of m homogeneous resources connected by any interconnection topology and the jobs arrive over time (*i.e.* in an on-line manner) in a centralized waiting queue. A *job* j is some workload which has the following data:

- The estimated processing time $\tilde{p_j}$ of the job informed by the user;

- The actual processing time p_j of the job (only known after the job has been executed);
- The resource requirement of the job, measured as the number of processors q_j;
- The submit time r_j of the job (also called release date).

Although some data sets have additional information, the selected variables are available in most real workload traces, shared using the Standard Workload Format (SWF) [32]. Typically several simplifications about j are made under the perspective of the scheduler: parallel efficiency, interdependence, and computation and communication intensities are often ignored. Instead, j is seen as an independent "black box" that will require q_j resources for $\tilde{p}_j$ units of time.

There are many objective functions that can be considered in the on-line scheduling problem. As mentioned in Section 2.1.1, one reasonable expectation is that the waiting time of a job should be proportional to its processing time [25]. Hence, one can use the *bounded slowdown* objective function which is defined as follows for a job j:

$$bsld = \max\left(\frac{w_j + p_j}{\max(p_j, \tau)}, 1\right) \tag{3.1}$$

where w_j is the time that job j waited for execution (*i.e.* the time that j starts its execution minus the submit time r_j) and τ is a constant, with a typical value of 10 seconds, that prevents small jobs from having excessively large slowdown values, and thus giving more stability for the slowdown (see Section 2.1.1) metric. Similarly, we can define the *average bounded slowdown* which is the slowdown average over a sequence of jobs J:

$$\mathsf{AVEbsld}(J) = \frac{1}{|J|}\sum_{j \in J} \max\left(\frac{w_j + p_j}{\max(p_j, \tau)}, 1\right) \tag{3.2}$$

In this work, all scheduling evaluations are performed using this objective function. However, scheduling methodology proposed in this chapter can be applied with

other objective functions, such as the ones presented in Section 2.1.1, thus giving freedom for the HPC platform maintainer to adopt the objective function that best suits the performance requirements of his/her users. Following the scheduling notation presented in Section 2.1.1, in this chapter we are concerned about the on-line parallel job scheduling problem $P_m \mid size_j, r_j \mid$ AVEbsld with rigid jobs.

3.2.2 Simulation Scheme

We considered for simulation an HPC platform represented by an homogeneous cluster compounded by m resources (processors). We defined two sets of jobs to be executed, S and Q. Set S contains $|S|$ jobs that are executed in order of arrival at the beginning of the simulation and it acts as a warm-up workload. Jobs from Q, which are used to extract information about scheduling performance, start to arrive after all jobs from S arrived. This scheme provides a way to represent an initial resource state of the cluster before the arrival of jobs from Q. From both sets, the jobs characteristics are obtained from a trace generated with the Lublin and Feitelson [55] workload model. To conceive this model, several real HPC workload logs were considered to fit a generalized workload model, capable of representing not only the jobs' geometry (p_j and q_j), but as well the release date of jobs r_j, including peak periods. Another advantage of the workload model is that we can generate arbitrarily large traces. The job sets S and Q are generated in the following way: from a large enough trace N generated by the workload model, we randomly select a subtrace $M \subset N$ with size $|S| + |Q|$. The first $|S|$ jobs from M will belong to set S and the remaining $|Q|$ jobs from M will belong to set Q.

The first step is to determine, using simulations, how the scheduling performance is affected when a job j is selected for processing, under different sets of jobs and resource states. To obtain this information, several tuples of job sets (S, Q) were generated, where each tuple (S, Q) is a randomly generated trace as aforementioned. The key idea is to list as many scheduling situations – represented by tuples (S, Q) – as possible. For each tuple (S, Q), we define $\mathcal{P}$ as a collection of random permutations of Q, and $\mathcal{P}(j_0 = j)$ as the subset from all permutations where j is the first job in

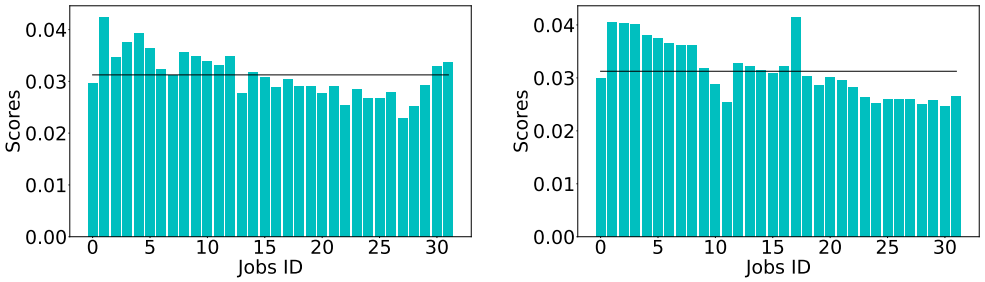
the permutation. We then simulate the scheduling execution for each pair (S, per) for all $per \in \mathcal{P}$. We call these pairs (S, per) as *trials* of the tuple (S, Q). On each trial, each job $j \in Q$ is submitted for execution in the order as they appear in per. We then assign a *score* for each job $j \in Q$:

$$score(j) = \frac{\sum_{per_j \in \mathcal{P}(j_0=j)} AVEbsld(per_j)}{\sum_{per_k \in \mathcal{P}} AVEbsld(per_k)} \tag{3.3}$$

The score denotes the impact of assigning a job $j \in Q$ to execute first, in the average bounded slowdown of all jobs in Q. Assuming that for each permutation $per \in \mathcal{P}$, the job j_0 is selected according to a uniform probability distribution, if a job j had a good impact in performance, then the sum present in the dividend of Equation 3.3 will be lower, when compared to another job that had a worse impact in the performance, and the value of $score(j)$ will be lower as well. The set of scores for all jobs $j \in Q$ constitute a *trial score distribution* of the jobs of Q under initial resource state S. Typical distributions, shown in Figure 3.1, contains most scores slightly above or below the mean $\frac{1}{|Q|} = \frac{1}{32} = 0.031$. Jobs with lower scores have a more positive impact in the average bounded slowdown when they are chosen to be executed first. It is important to note that, as the total number of possible trials grows quickly in function of $|Q|$, the size of the set Q must be chosen in a way that it is possible to perform an accurate estimation of the jobs' scores, while maintaining a feasible computational cost. More information about this accuracy and computational cost can be found in Section 3.3.1.

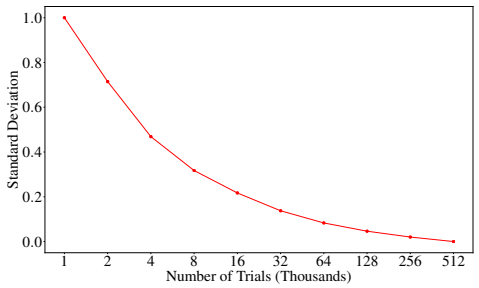
By joining the generated samples from multiple distinct tuples (S, Q), we generate a distribution $score(p, q, r)$, containing the sample means for jobs with processing time p, number of used processors q and arrival time r. This is the central result obtained from the simulations. The idea is that a scheduler from a HPC system will select the job from the queue with (p, q, r) values that has the smallest $score(p, q, r)$ value.

Using simulations to observe the scheduling behavior is rather adopted in the literature, with some works [53, 67] using the simulation results directly in order to forecast a better scheduling strategy. In the next section we present a novel approach



(a) One example of a trial score distribution

(b) Another example of a trial score distribution



(c) Normalized standard deviations for the trial score distributions obtained with different numbers of the trials

Figure 3.1 Examples of trial score distributions generated by the simulation procedure for a tuple of job sets (S, Q), with $|S| = 16$ and $|Q| = 32$, in a cluster with 256 nodes, and their estimation accuracy in function of the number of trials. The black horizontal in (a) and (b) line represents the mean $\frac{1}{|Q|} = \frac{1}{32} = 0.031$.

that, rather than using the simulation results directly to perform scheduling decisions, it uses the simulation results to learn and create a nonlinear function, that can be afterwards used as an on-line scheduling policy.

3.2.3 Machine Learning Scheme

Using simulations we generate many irregular $score(p, q, r)$ distributions, with values that can change each time a simulation is performed and that provide good estimates only for certain jobs characteristics. To obtain smoother and more general representations of the score distributions, we can use a machine learning technique, called nonlinear regression, to determine a nonlinear function $f(p, q, r)$ that provides a good fitting to the distribution. This function can then be later assigned as a scheduling policy. In other words, the jobs arriving into a centralized queue of an HPC system can be prioritized according to these functions.

Let J be the set of all jobs from all sets Q generated in the simulation phase (see Section 3.2.2). For a job $j \in J$, we have a 4-tuple $(p_j, q_j, r_j, score(p_j, q_j, r_j))$ obtained

from the previously computed trial score distributions. This 4-tuple denotes the *observation* of the scheduling performance behavior of the job j. Given a collection $\mathcal{F}$ of nonlinear functions, the problem consists in finding a function $f(p_j, q_j, r_j) \in \mathcal{F}$ that better fits the distribution $score(p, q, r)$ generated from all jobs $j \in J$.

We defined $\mathcal{F}$ as all functions of form $y(p_j, q_j, r_j, \theta) = (\theta_1\phi_1(p_j))\ op_1\ (\theta_2\phi_2(q_j))\ op_2\ (\theta_3\phi_3(r_j))$. We call ϕ_1, ϕ_2 and ϕ_3 as *base functions* and they can be any of the functions: identity, square root, logarithm and inverse. Operators op_1 and op_2 are any of the operators sum (+), multiplication ($\cdot$) or division ($\div$). Coefficients θ_1, θ_2 and θ_3 denotes the relative importance of the base functions and are obtained by a nonlinear regression fitting.

The idea behind this scheme is to explore a wide variety of relationships – represented by the many functions that can be obtained with the functional form $y(p_j, q_j, r_j, \theta)$ – between the jobs characteristics, with the hypothesis that there may be one or more functions $f(p_j, q_j, r_j, \theta) \in \mathcal{F}$ that, after a regression fitting, will stand out as an efficient scheduling policy. It is important to note that the relationship between the parameters θ_1, θ_2 and θ_3 can be nonlinear, thus requiring a nonlinear regression to perform the fit (see Section 2.2.1). We set models with only three parameters in order to maintain a reasonable number of functions $f(p_j, q_j, r_j, \theta) \in \mathcal{F}$ to perform the fitting. Also, by using few parameters we reduce the possibility of overfitting (see Section 2.2.1). We employ a weighted nonlinear regression [12] procedure, which minimizes the weighted sum of squared loss function:

$$\Sigma_{wL} = \sum_{j \in J} ((p_j q_j) \cdot (f(p_j, q_j, r_j, \theta) - score(p_j, q_j, r_j)))^2 \tag{3.4}$$

The above equation is similar to the sum of squared loss (Equation 2.12), with the exception that we used the weight $(p_j q_j)$ to emphasize that the fit must perform a good estimation of the score of large area jobs (*i.e.* jobs with large p and q values). This is based on the argument that jobs that consume a large amount of resources for a long period of time have a potential of blocking the execution of many smaller jobs, degrading the overall scheduling performance.

Once we perform the nonlinear regression fitting for all functions $f(p_j, q_j, r_j, \hat{\theta}^f) \in \mathcal{F}$, where $\hat{\theta}^f$ are the coefficients found during the nonlinear regression fit for $f(p_j, q_j, r_j, \theta)$, we use a *Mean Absolute Error* function (*MAE*, Equation 3.5, see Section 3.2.3) to evaluate the average error of each nonlinear function $f(p_j, q_j, r_j, \hat{\theta}^f)$, when used to estimate the $score(p_j, q_j, r_j)$ for all jobs $j \in J$. The hypothesis is that nonlinear functions with the lower *MAE* values would perform well as scheduling policies.

$$MAE(f) = \frac{1}{|J|} \sum_{j \in J} \| f(p_j, q_j, r_j, \hat{\theta}^f) - score(p_j, q_j, r_j) \| \tag{3.5}$$

3.3 Results

In this section we present the main results obtained of the experiments. We first describe the simulation procedure and the nonlinear functions obtained with machine learning. Next, we evaluate the performance of the obtained functions when scheduling synthetic workloads under different conditions: (i) using actual job processing times, (ii) using user estimated job processing times, and (iii) using user estimated job times with backfilling. Finally, we evaluated the obtained functions using real workload traces, for the same three conditions used with the synthetic workloads.

In the simulations, to generate the distribution $score(p, q, r)$, we considered an HPC platform compounded by $m = 256$ homogeneous processors. We used sets of jobs S and Q with $|S| = 16$ and $|Q| = 32$ jobs. All simulations were performed using SimGrid [13].

We compared the performance of our scheduling policies with a selection of classical scheduling and smart hand-engineered policies, used in real HPC platforms (Table 3.1). Two classical and well known policies are First Come First Served (FCFS), where jobs are scheduled by the arrival order, and Shortest Processing Time First (SPF), where jobs with smaller processing times are scheduled first. We also used

Table 3.1 Scheduling policies used for comparison.

Name	Function
FCFS [70]	$score(j) = r_j$
SPF [71]	$score(j) = p_j$
WFP3	$score(j) = -(w_j/p_j)^3 \cdot q_j$
UNICEF	$score(j) = -w_j/(\log_2(q_j) \cdot p_j)$

the WFP3 (WFP, for short) and UNICEF (UNI, for short) policies [75], which are based on the processing time (p_j), requested number of processors (q_j), and waiting time (w_j) of the job. The intuition behind WFP is that shorter and/or older jobs should be largely favored, while preventing the starvation of large jobs. UNI in its turn attempts to provide a fast turnaround for small jobs by favoring them.

3.3.1 Machine Learning: Obtained Nonlinear Functions

The first step towards obtaining the nonlinear functions for usage as scheduling policies is to produce the distribution $score(p, q, r)$. We start by generating permutations of the set of jobs Q, which are used to construct the trial score distributions. Enumerating and simulating the execution of all permutations of a set of jobs of size $|Q| = 32$ is computationally prohibitive and, therefore, we need to define a suitable number of permutations that generate accurate trial score distributions. For that, we selected one tuple (S, Q) and generated the trial distributions with increasing amount of trials, repeating the simulation procedure ten times per number of trials, and measuring the standard deviation of the estimated scores. Figure 3.1c shows that the standard deviation drops quickly with increasing amount of trials. With 256 thousand trials, the resulting normalized standard deviation was 0.02. We decided to use 256 thousand trials, since its simulation takes less than 11 minutes using SimGrid [13] on an Intel Xeon E5-2620v2 six-core CPU.

After obtaining the trial score distributions, we generated the distribution $score(p, q, r)$ and performed the nonlinear regression using the function `leastsq()` from the SciPy [47] Python library for all functions $f(p_j, q_j, r_j, \theta) \in \mathcal{F}$. Table 3.2 shows the four best functions obtained with regard to the Equation 3.5. We mathematically

simplified the obtained functions, merging the coefficients θ_1, θ_2 and θ_3 into a single coefficient $\theta_3/(\theta_1\theta_2)$, in front of the $log(r_j)$ term.

A noticeable phenomenon is their similarity, with all functions constituted by a sum of two factors, one containing parameters p_j and q_j and the other with the dependence on $log(r_j)$. Considering the large values of the constant before the $log(r_j)$ term, the functions emphasize that jobs that arrived earlier (*i.e.* with lower r_j values) must be prioritized in order to maintain lower slowdowns (recall that jobs with lower score value have a high priority). Moreover, these large constants effectively prevent starvation without any manual customization of the policies. Figures 3.2b and 3.2c illustrate the strong dependency on the submission time for all policies F1 to F4, with jobs that arrive earlier receiving a large priority (darker colors) over more recent jobs.

The second important factor is the size of the job, a product of two functions $\alpha(p_j)$ and $\beta(q_j)$ of the processing time and number of processors used by the jobs. The policies F1 to F4 differ in the relative importance given to each of these values (p_j and q_j), with F1 and F2 imposing a heavier penalization for increasing numbers of requested processors q_j, F4 penalizing larger processing times p_j, and F3 penalizing higher values of p_j and q_j equally. Figure 3.2a shows that, for a fixed value of r_j, higher priorities are given for jobs that have either required smaller processing times or number of processors.

These results comply with the general intuition – in which jobs with small processing time, small requested amount of processors and that were submitted earlier should be prioritized – that is adopted by most of the hand-engineered scheduling policies. The main differences are the adoption of two separate terms, one considering only job size and the other submission time, and the large coefficient before the submission time term.

3.3.2 Scheduling Performance: Workload Model

In this subsection we aim to answer the following question: *Can the nonlinear functions, obtained using the procedure from Section 3.3.1, perform well as scheduling*

Table 3.2 The four best nonlinear functions obtained using nonlinear regression.

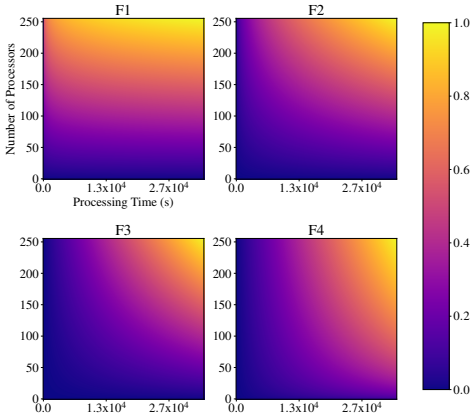
ID	Nonlinear Function	*MAE(f)*
F1	$\log_{10}(p_j) \cdot q_j + 8.70 \cdot 10^2 \cdot \log_{10}(r_j)$	$5.278 \cdot 10^{-3}$
F2	$\sqrt{p_j} \cdot q_j + 2.56 \cdot 10^4 \cdot \log_{10}(r_j)$	$5.317 \cdot 10^{-3}$
F3	$p_j \cdot q_j + 6.86 \cdot 10^6 \cdot \log_{10}(r_j)$	$5.408 \cdot 10^{-3}$
F4	$p_j \cdot \sqrt{q_j} + 5.30 \cdot 10^5 \cdot \log_{10}(r_j)$	$5.482 \cdot 10^{-3}$

policies for jobs generated by the Lublin and Feitelson [55] workload model in the following scenarios?

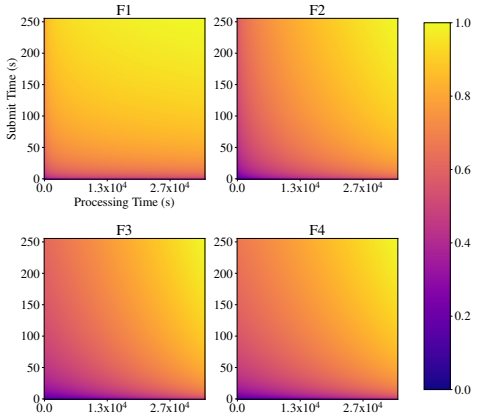
- Using the actual processing time p_j in the scheduling decisions and the same number of processors $m = 256$ from the simulation scheme;
- Using the actual processing time p_j in the scheduling decisions, but increasing the number of processors to $m = 1024$;
- Using the processing time estimate $\tilde{p}_j$ provided by the user, instead of the actual processing time p_j, to perform the scheduling decisions;
- Using the processing time estimate $\tilde{p}_j$ provided by the user, but performing the scheduling using the aggressive backfilling algorithm.

We should emphasize that in all scenarios we used the same set of nonlinear functions, obtained using the actual processing time p_j of jobs and $m = 256$ processors. Since the functions are parametrized by the number of processors (q_j), submission time (r_j) and processing time p_j (which can be substituted by its estimate $\tilde{p}_j$), they can be used in different scenarios. The objective was to evaluate the generalization capabilities of the obtained nonlinear functions.

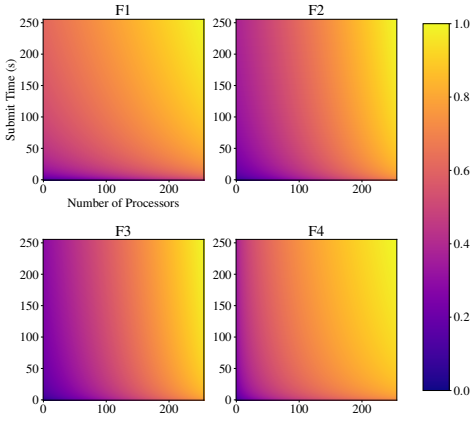
For all experiments, we define a *dynamic scheduling experiment* as being multiple simulations of the execution of a sequence of jobs obtained from a certain workload trace or model. For each simulation we choose one scheduling policy from Tables 3.1 and 3.2 to schedule all jobs in the sequence. The output of the dynamic scheduling experiment is the average bounded slowdown from each simulation performed, using all policies present in Tables 3.1 and 3.2. The sequence of jobs contains all jobs submissions over a period of fifteen days. When using workload traces, we assured



(a) Processing time *versus* number of processors



(b) Processing time *versus* submit time. Older jobs have small submit time values



(c) Number of processors *versus* submit time. Older jobs have small submit time values

Figure 3.2 Dependency on the parameters p_j, q_j and r_j for the four best nonlinear functions obtained.

that there was no overlap between sequences used on distinct dynamic scheduling experiments.

The simulated on-line scheduling algorithm works as follows: jobs arrive in a centralized waiting queue and the scheduler performs a reschedule – using a scheduling policy – of the jobs present in this queue in two distinct events: (i) when a job arrives in the queue or (ii) when a resource (set of processors) is released and becomes available. When a job j is selected for execution and if the requested number of processors q_j is lower than the total number of processors available, then q_j processors are reserved for this job and they become unavailable. These processors will become available again only when p_j units of time have passed since the start of the

Table 3.3 Median of the average bounded slowdowns from Subsections 3.3.2 and 3.3.3.

Experiment	FCFS	WFP	UNI	SPF	F4	F3	F2	F1
Workload model, $m = 256$, actual runtimes p_j	7830.58	4018.25	2717.84	1222.67	368.37	151.53	**37.47**	42.88
Workload model, $m = 256$, runtime estimates $\tilde{p}_j$	7830.58	7505.74	4265.58	5841.89	590.40	306.68	60.76	**50.67**
Workload model, $m = 256$, aggressive backfilling	1306.38	1220.52	508.95	863.76	122.72	54.49	**42.91**	50.77
Workload model, $m = 1024$, actual runtimes p_j	6910.82	5229.28	3176.86	2404.65	514.43	73.43	**21.41**	24.78
Workload model, $m = 1024$, runtime estimates $\tilde{p}_j$	6910.82	6564.50	3790.40	5220.45	1146.34	559.86	123.10	**32.33**
Workload model, $m = 1024$, aggressive backfilling	1262.63	1651.83	1079.08	1296.65	377.99	180.75	44.70	**30.24**
Curie workload trace, actual runtimes p_j	290.27	206.57	110.14	139.58	18.77	8.21	**4.27**	6.57
Anl Interpid workload trace, actual runtimes p_j	40.93	11.06	5.83	3.91	2.12	**1.76**	2.07	2.15
HPC2N workload trace, actual runtimes p_j	90.61	22.20	11.37	12.16	11.65	15.40	**10.47**	12.12
SDSC Blue workload trace, actual runtimes p_j	370.22	67.83	29.53	24.65	20.03	12.63	**4.41**	8.22
SDSC SP2 workload trace, actual runtimes p_j	629.60	59.60	25.84	21.78	29.14	45.33	**15.58**	19.53
CTC SP2 workload trace, actual runtimes p_j	438.37	309.72	33.86	90.24	18.77	15.21	**5.32**	9.13
Curie workload trace, runtime estimates $\tilde{p}_j$	290.27	258.17	145.19	218.77	44.63	12.15	**8.77**	9.86
Anl Interpid workload trace, runtime estimates $\tilde{p}_j$	40.93	18.33	11.94	9.17	4.18	3.29	**2.63**	2.74
HPC2N workload trace, runtime estimates $\tilde{p}_j$	90.61	62.24	28.34	46.67	27.81	20.08	**16.90**	19.83
SDSC Blue workload trace, runtime estimates $\tilde{p}_j$	370.22	136.11	57.00	46.82	18.69	9.66	**8.20**	11.52
SDSC SP2 workload trace, runtime estimates $\tilde{p}_j$	629.60	243.86	91.57	48.43	40.21	41.38	**30.04**	34.13
CTC SP2 workload trace, runtime estimates $\tilde{p}_j$	438.37	369.93	104.98	306.78	21.76	18.23	13.46	**12.33**
Curie workload trace, aggressive backfilling	54.94	49.23	32.27	35.72	14.90	9.57	**7.80**	9.81
Anl Interpid workload trace, aggressive backfilling	9.66	7.19	4.39	3.79	3.59	2.87	**2.65**	2.74
HPC2N workload trace, aggressive backfilling	36.43	28.08	16.22	16.26	15.31	**15.26**	16.83	20.46
SDSC Blue workload trace, aggressive backfilling	36.40	14.07	11.08	9.87	8.11	**8.02**	8.52	11.59
SDSC SP2 workload trace, aggressive backfilling	66.38	41.61	32.66	28.38	26.68	25.31	**23.26**	32.06
CTC SP2 workload trace, aggressive backfilling	83.38	54.32	24.96	14.40	**9.90**	11.07	10.23	12.88

execution of j. If there is not enough processors to process j, then the scheduler waits for one of the two rescheduling events mentioned above.

3.3.2.1 Scheduling using actual job runtimes p_j

In this experiment we generated a job queue with characteristics from the workload model of Lublin and Feitelson [55] and considering a HPC platform constituted by $m = 256$ processors, which are the same settings used in the simulations to generate the nonlinear functions. Figure 3.3a shows the average bounded slowdowns for fifty dynamic scheduling experiments, with the orange line representing the median of the average bounded slowdown (whose values are shown in Table 3.3) over these experiments, the box limits representing the upper and lower quartiles, and the whiskers representing the lowest and highest values outside the the box limits but still inside the range of 1.5 times the difference between the upper and lower quartiles. The figure shows that all obtained nonlinear functions performed substantially better than the evaluated existing scheduling policies, from Table 3.1. The nonlinear function F2 provided the best results, followed by functions F1, F3 and F4. Although we would expect F1 to perform better, the *MAE* values of the generated functions are

similar (Table 3.2) and, consequently, we cannot assume that F1 will always provide the best results.

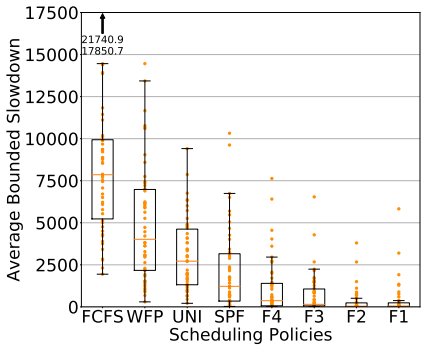
Figure 3.3d shows the results of fifty dynamic scheduling experiments considering an HPC platform with $m = 1024$ processors. The jobs were generated using the Lublin and Feitelson workload model configured for a cluster with 1024 processors, so that jobs sent to the waiting queue would use between 1 and 1024 processors. We can see that, compared to the other scheduling policies, the obtained nonlinear functions continued to perform well, indicating that the obtained scheduling policies have some generalization capability regarding the number of processors in the HPC platform.

3.3.2.2 Scheduling using user estimated job runtimes

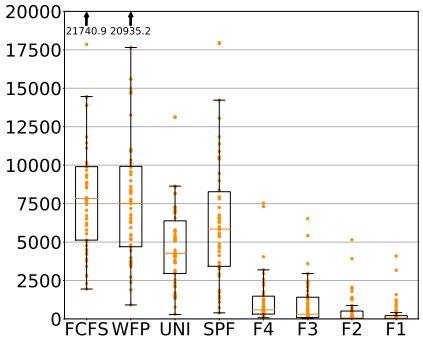
In this experiment, instead of using the processing time p_j in the scheduling decisions, we utilize the *estimated* processing time $\tilde{p}_j$ of the job that is provided by the user. The actual processing time p_j in this case is used only to simulate the execution of the job. For the workload model used in this work, we used the user runtime estimate model of Tsafrir *et al.* [78] to generate $\tilde{p}_j$.

Traditionally the processing time estimates provided by the user are highly inaccurate. In this light, it is expected a reduction in the scheduling performance of all the scheduling policies, since none of these scheduling policies are designed to handle inaccuracies in the execution time of the jobs, and therefore the only aspect that we can evaluate is how tolerant the scheduling policies are when inaccurate processing time estimates are introduced.

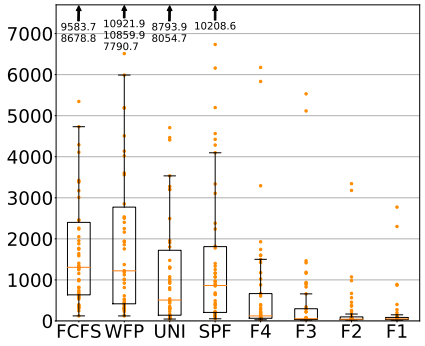
Figures 3.3b and 3.3e show the results of fifty dynamic scheduling experiments, using estimated processing times $\tilde{p}_j$, for HPC platforms constituted by $m = 256$ and $m = 1024$ processors. As expected, all scheduling policies had a considerable performance degradation, except the FCFS, which does not use job processing times. Nevertheless, the median of the average bounded slowdown generated by policies F1, F2, F3 and F4 was between 7.22 and 84.18 times better for the scenario with



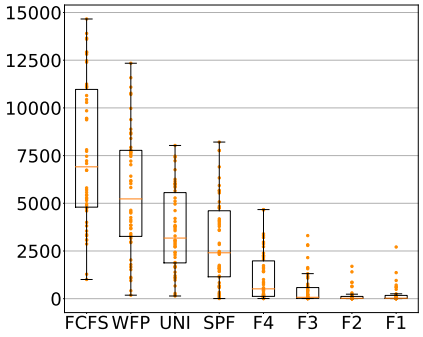
(a) jobs characteristics generated from the Lublin and Feitelson [55] model and $m = 256$



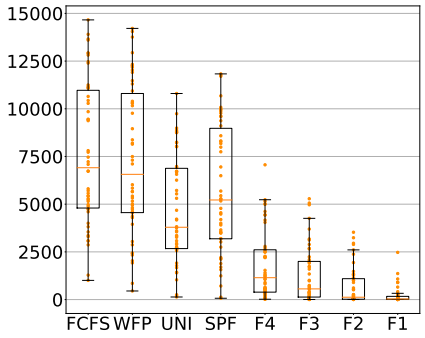
(b) jobs characteristics generated from the Lublin and Feitelson [55] model, with processing time estimates generated from the Tsafrir *et al.* [78] model and $m = 256$



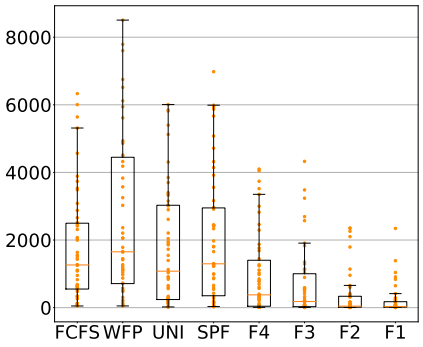
(c) jobs characteristics generated from the Lublin and Feitelson [55] model, with processing time estimates generated from the Tsafrir *et al.* [78] model and $m = 256$ and with aggressive backfilling



(d) Same settings as (a), however with $m = 1024$



(e) Same settings as (b), however with $m = 1024$



(f) Same settings as (c), however with $m = 1024$

Figure 3.3 Scheduling performance results with jobs characteristics generated from a workload model.

$m = 256$ and between 3.30 and 117.24 times better for the scenario with $m = 1024$, when compared to the best performing hand-engineered scheduling policy.

3.3.2.3 Scheduling using estimated runtimes and agressive backfilling

In this experiment, we used the aggressive backfilling algorithm in conjunction with the scheduling policies. In such setting, when a rescheduling event occurs, the jobs are reordered in the queue using a scheduling policy and then we apply the aggressive backfilling algorithm to check if there is one or more jobs further back in the queue that, if selected for execution, will not delay the first job in the queue. In this case, all the scheduling decisions (the scheduling policy and the backfilling) are made over the estimated processing time $\tilde{p}_j$, with the actual processing time p_j used only to simulate job execution. This setting is the most realistic setting we can elaborate using jobs generated from a workload model.

Figures 3.3c and 3.3f show that the introduction of the aggressive backfilling algorithm resulted in a overall increase in performance, with the FCFS policy with backfilling (previously mentioned as the EASY algorithm) taking the most advantage of the backfilling strategy. Our obtained nonlinear functions had the least benefits from the backfilling, since the better initial schedules lowered the possibilities for job backfilling. Nevertheless, the performance of the obtained nonlinear functions is still superior to the other hand-engineered scheduling policies. For instance, the median average slowdown for the F2 strategy was more than 11 times smaller than the best hand-engineered policy for both 256 and 1024 core machines.

3.3.3 Scheduling Performance: Real Workload Traces

We also evaluated whether the obtained scheduling policies generalize well to highly different workloads types, obtained from real workload traces, and HPC platform configurations. In this light, in this subsection we attempt to answer the following questions:

- Can the obtained scheduling policies perform well when scheduling a set of jobs extracted from real workload traces and executed in a simulated HPC platform similar to the one where the traces were obtained?

Table 3.4 Real workload traces used for evaluation of the scheduling policies.

Name	Year	# CPUs	# Jobs	Util %	Duration
Curie	2011	93,312	312,826	62.0	20 Months
ANL Interpid	2009	163,840	68,936	59.6	8 Months
HPC2N	2002	240	202,871	60.1	42 Months
SDSC Blue	2000	1,152	243,306	76.7	32 Months
SDSC SP2	1998	128	59,715	83.4	24 Months
CTC SP2	1997	338	77,222	85.2	11 Months

- With the same setting from the previous question, but using the user estimated processing times $\tilde{p}_j$ to perform the scheduling decisions, can the obtained nonlinear functions perform well as scheduling policies?
- Can the obtained nonlinear functions benefit from the aggressive backfilling algorithm and perform well in the scenario from the previous question?

We used the traces described in Table 3.4, which are publicly available at the Parallel Workloads Archive [32]. To better evaluate the generality of the obtained nonlinear functions, we chose a set of traces from computer HPC configurations ranging from 128 to 163,840 processors, mean utilization values from 59.6% to to 85.2% and measurements dates from year 1997 to 2011.

On each trace we collected as many non overlapping half-month sequences of jobs to perform the dynamic scheduling experiments as possible. Each sequence contains all jobs submissions equivalent to a period of fifteen days. We made sure that there was no overlap between the sequences and the on-line scheduling algorithm works similarly to the scheduling algorithm used in the experiments of the previous subsection.

3.3.3.1 Scheduling using the actual job runtimes p_j

Figure 3.4 shows the dynamic scheduling experiments results using the actual processing times to perform the scheduling decisions. All the obtained nonlinear functions resulted in lower average slowdowns for all traces, with varying levels of improvements depending on the HPC workload characteristics. More importantly, the difference between the upper and lower quartiles (box limits) was lower when

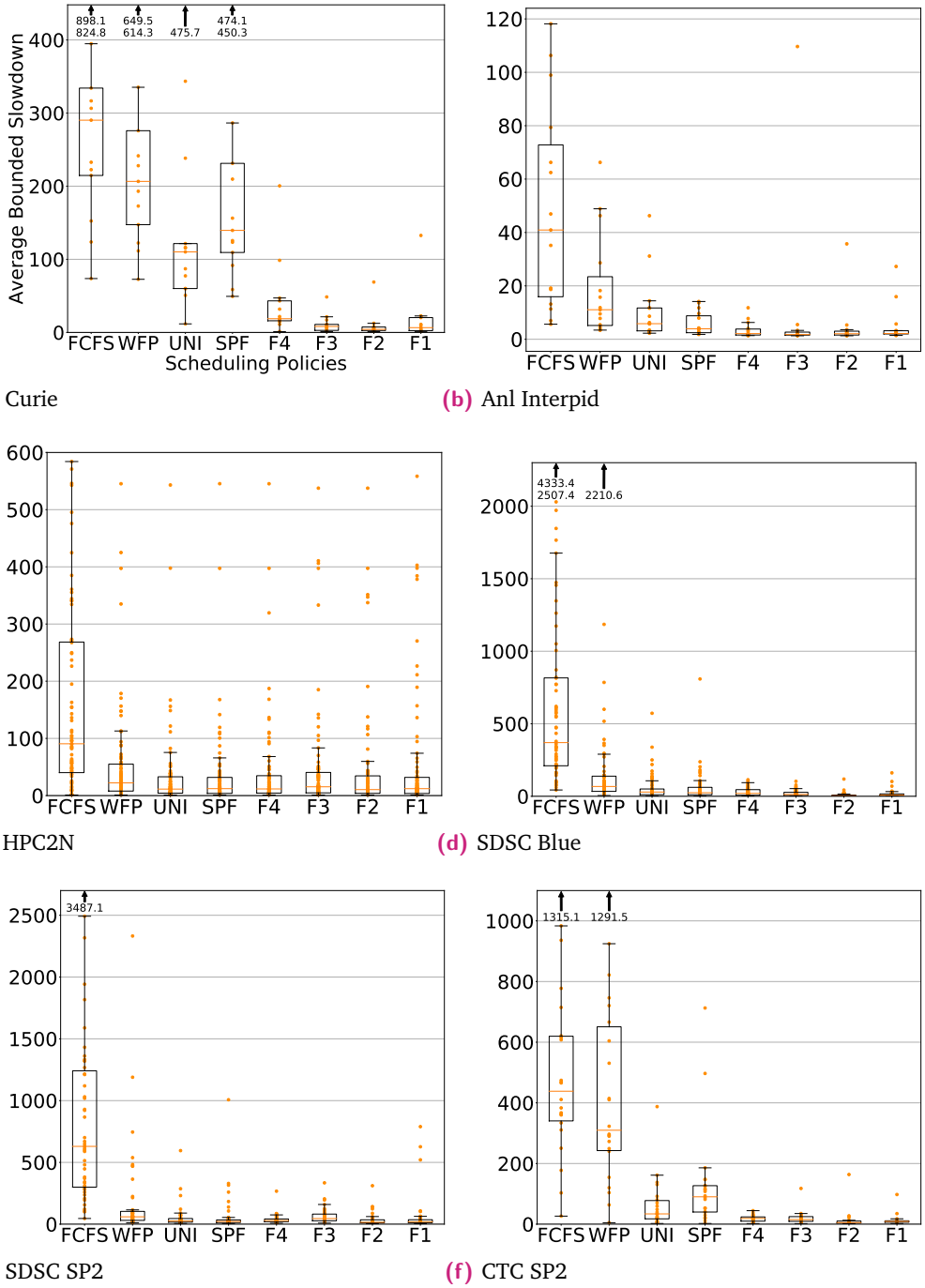
using the obtained nonlinear functions, meaning that the average slowdown was more predictable and stable, a desirable property for HPC systems. For all policies there were some cases where the average bounded slowdown was notoriously high, which occurred due to uncommon jobs characteristics in the traces. For example, in the HPC2N trace, some workloads contains bursts of jobs with large processing times, which overloaded the platform for all policies.

The nonlinear function F2 achieved overall best results for most of the traces, with the exception of the the ANL Interpid trace, in which F3 performed better. This result shows that, with real workload traces, the best scheduling policy can change from trace to trace. This behavior is not unexpected, since the workload traces evaluated are different from each other and from the workload generation model used in the simulation scheme. But choosing any of the obtained policies F1 to F4 resulted in improvements in the median average slowdowns in all scenarios and smaller differences between the upper and lower quartiles in most scenarios.

3.3.3.2 Scheduling using user estimated job runtimes $\tilde{p}_j$

We evaluated the scheduling policies using the processing time estimate $\tilde{p}_j$ obtained from the respective workload log when performing the scheduling decisions. Since the user estimates of the processing times are often rough and inaccurate, we expect a degradation in the performance of all scheduling policies. Figure 3.5 shows that the obtained functions F1 to F4 continued to generate lower median average slowdowns and differences between the upper and lower quartiles for all evaluated HPC platforms. Although the best function from F1 to F4 varied depending on the platform, any of them would result in performance improvements over existing hand-engineered policies.

The results from this section are noteworthy, considering the nonlinear functions F1 to F4 were trained using data from a single workload model in a simulated machine with 256 processors. These functions worked well as scheduling policies for HPC machines with highly different architectures, with up to 163,840 processors, very



(a) Curie (b) Anl Interpid

(c) HPC2N (d) SDSC Blue

(e) SDSC SP2 (f) CTC SP2

Figure 3.4 Scheduling performance results with jobs characteristics obtained from real HPC platform workload logs and using actual processing time in the scheduling decisions.

different workload types, and using inaccurate estimated job runtimes from real machine users.

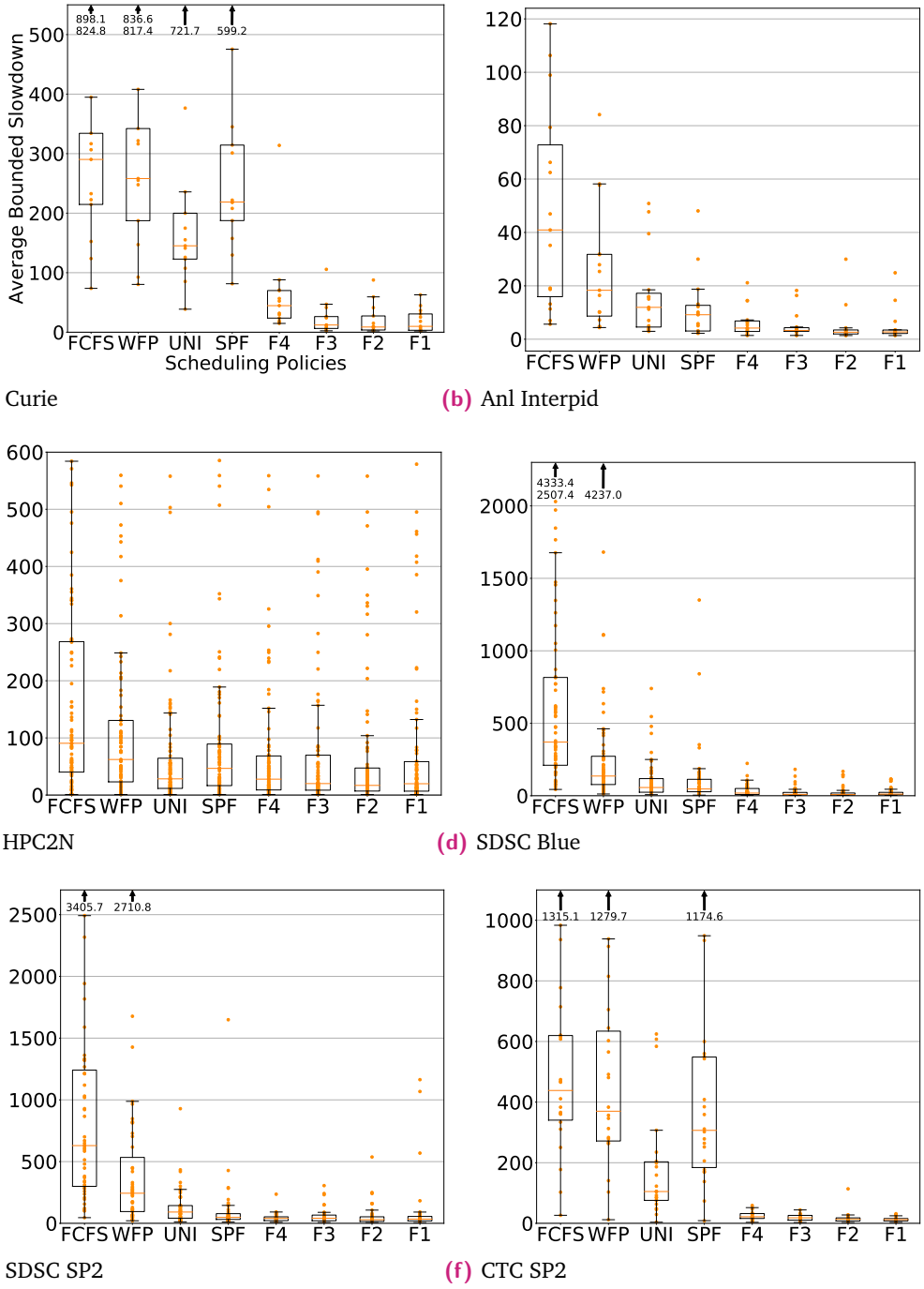
3.3.3.3 Scheduling using estimated processing times and aggressive backfilling

In this experiment we considered the most realist scenario, where scheduling decisions are based on the user estimates of processing times $\tilde{p}_j$ and with the addition of aggressive backfilling to reduce resource idleness. Figure 3.6 shows the corresponding dynamic scheduling experiments. Once again, the FCFS policy with backfilling (the EASY scheduling algorithm) was the scheduling policy which benefited the most with the introduction of backfilling. The performance results obtained for the WFP and UNI policies also reinforces the favorable results obtained by Tang *et al.* [75], since we obtained similar comparative results for these policies, with the exception that we evaluated them with different workload logs.

The obtained nonlinear functions had smaller benefits from using backfilling. Similarly to the experiments with synthetic workloads, the better schedules provided by the proposed nonlinear functions resulted in less opportunities for backfilling jobs. Nevertheless, functions F1 to F4 still resulted in lower median average slowdowns and/or lower differences between the extreme quartiles for most scenarios, and continued to be a better general choice than the hand-engineered scheduling policies.

3.4 Summary

Due to its simplicity, scheduling policies in the form of functions that take job characteristics into consideration, combined with the computationally inexpensive aggressive backfilling algorithm, are an appealing alternative for the problem of online scheduling of jobs in HPC platforms. In an equally simple manner, in this chapter we show that – by introducing a simulation procedure that captures the observations of job scheduling behavior under several distinct conditions and a simple machine learning strategy to model these observations into nonlinear functions – we can obtain scheduling policies that effectively capture the effects of job characteristics on scheduling behavior. We showed that these functions outperform other classical



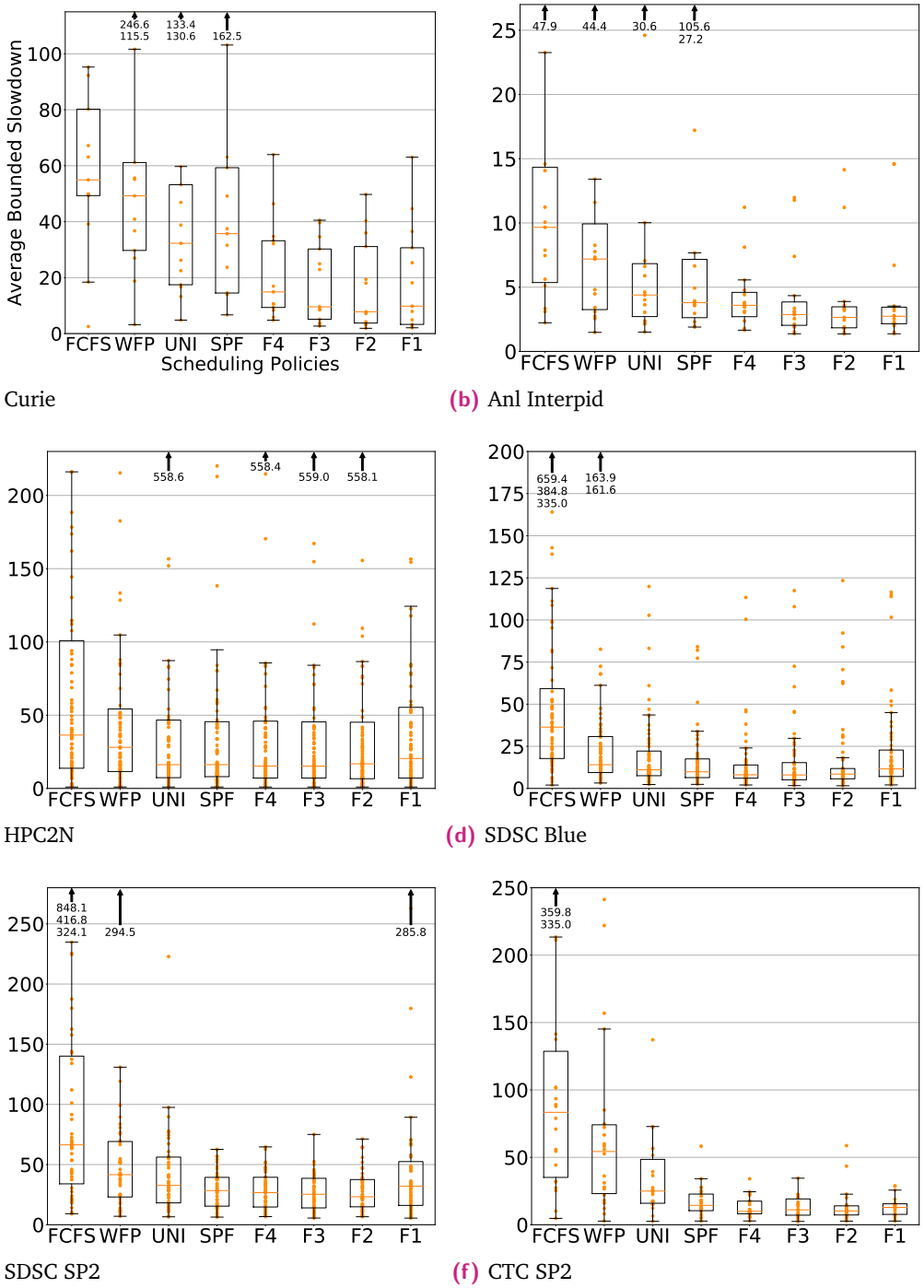
(a) Curie (b) Anl Interpid

(c) HPC2N (d) SDSC Blue

(e) SDSC SP2 (f) CTC SP2

Figure 3.5 Scheduling performance results with jobs characteristics obtained from real HPC platform workload logs and using user estimated processing times obtained from the same logs in the scheduling decisions.

and smart hand-engineered scheduling policies in a variety of scenarios. Moreover, the large weights assigned to the submission time of the jobs (Table 3.2) result in a fast increase in priority as the jobs wait in the queue (Figures 3.2b and 3.2c), thus preventing starvation without any manual customization of the policies.



(a) Curie **(b)** Anl Interpid

(c) HPC2N **(d)** SDSC Blue

(e) SDSC SP2 **(f)** CTC SP2

Figure 3.6 Scheduling performance results with jobs characteristics obtained from real HPC platforms, with the addition of the aggressive backfilling, and using user estimated processing times obtained from the same logs in the scheduling decisions.

Using jobs characteristics obtained from the workload model of Lublin and Feitelson [55] and considering similar scenarios from the one used to capture the scheduling observations, the obtained scheduling policies achieved very good results. In the most realistic of these scenarios (*i.e.* using the aggressive backfilling in addi-

tion to the scheduling policies and using the processing time estimates to perform the scheduling decisions), the obtained scheduling policies achieved medians of the average bounded slowdowns up to 11.8 times better when compared to the best performing hand-engineered scheduling policy.

Although we used a workload model to generate the nonlinear functions, we could envision the same procedure being applied to obtain custom scheduling policies for a specific HPC platform, using its specific workload traces and architecture configurations. Our results using the workload model indicate that these custom policies could, in principle, bring important improvements in the achieved average bounded slowdowns in these platforms. We also have confidence that the procedures presented in this chapter can be applied with other objective functions such as average waiting time or flow time. Therefore, our approach gives freedom for the HPC platform maintainer to choose the objective function that best suits the requirements of his/her users. Also, one can apply the proposed procedure periodically, using data from newly executed jobs. This dynamic update will adapt the scheduling policies to changes in the pattern of jobs submitted in the platform.

Scheduling performance could also be improved by combining our machine learning scheme for obtaining efficient scheduling policies with a machine learning scheme for predicting job execution times, such as the one proposed by Gaussier *et al.* [40]. We showed that inaccurate execution time estimates from users greatly reduces the efficiency of our scheduling policies. We believe that by using execution time predictions we could improve the average bounded slowdown in the more realistic scenario where we do not know the actual job execution times.

Finally, we would like to note that using a workload model to capture the observations brought some important advantages. The generalized job properties present in the Lublin and Feitelson [55] workload model, when used to observe the scheduling behavior of the jobs in several distinct configurations, resulted in scheduling policies that are able to express efficient general patterns regarding to which job should be selected for execution first. In this light, the obtained scheduling policies showed consistent lower median values for the average bounded slowdown in simulation

experiments considering very different job types and HPC platform configurations. Therefore, although it would be possible to define scheduling policies specifically for each HPC platform, as aforementioned stated, it seems that general policies, as the ones we found in this work, could be sufficient to efficiently schedule jobs for a range of specific workload types and HPC platform configurations.

As future work, we can improve the work presented in this chapter in many directions. A first direction would be to perform a deep analysis of the schedules performed by the simulation scheme presented in Section 3.2.2. The simulation output itself can provide useful insights about good scheduling strategies that could result in novel scheduling heuristics.

It is equally interesting to perform the proposed approach to learn scheduling policies taking into account not only synthetic jobs, but as well jobs obtained in real HPC platform workload traces, such as the ones presented in Table 3.4. We envision that we could extract novel characteristics – as well as novel scheduling heuristics – of each HPC platform by performing our proposed approach in each workload trace.

Another direction would be to evaluate the effectiveness of the proposed approach with different functional forms $y(p_j, q_j, r_j, \theta)$ (see Section 3.2.3), more specifically polynomials of the jobs characteristics, and nonlinear variants of the area property of the jobs, notably $f(p_j, q_j, \theta) = p_j^{\theta_1} q_j^{\theta_2}$.

It would be interesting as well to evaluate the scheduling policies by deploying them on real HPC platforms and check how they perform under real conditions. We used simplified simulations to evaluate the obtained scheduling policies using real traces which ignores, for instance, network and memory bottlenecks that could appear from interactions among job execution. Nevertheless, we believe the simulations are a reasonable approximation for these platforms.

At last, another direction would be to cover a wider range of HPC platforms. We plan to improve the strategy proposed by this chapter to obtain scheduling policies that also address the on-line scheduling of jobs in HPC platforms containing processing

units with distinct architectures such as GPUs [62] and MICs [20] – thus characterizing a scheduling problem with unrelated machines in parallel R_m, see Section 2.1.1 – where multiple implementations, aiming a specific architecture, are available for the same job and the scheduler needs to select one of these implementations to be executed.

4

Beyond EASY-Backfilling: a Simple Scheduling Policies Case Study[2]

In the previous chapter we show that, using the rich information present in HPC workload logs to drive fast simulations that extract and log scheduling characteristics, and feeding these characteristics to simple non-linear regression machine learning models, we can devise scheduling heuristics that can perform well in several distinct scheduling scenarios. Although our results show sound advancements towards learning scheduling policies, one observation that we found is that there are some aspects of the learned scheduling policies that are hard to reason about. For instance, it is nontrivial to reason over the "magic" values present in the coefficients, as well as it is nontrivial to reason over the emergence of square roots and logarithms in the functions F1 to F4 (see Table 3.2). This lack of full explanation behind the scheduling decisions performed by functions F1 to F4 hinders the deployment of these learned scheduling policies in real scenarios, as many practitioners prefer to stay at their "comfort zone", and opt to use algorithms inspired by the Aggressive Backfilling with First-Come-First-Served order (also called EASY Backfilling [60], see Section 2.1.2.3).

At their defence, many reasons can be devised to justify the choice of EASY Backfilling: it is established that EASY Backfilling increases the overall utilization of the platform, while keeping a relative simplicity and job starvation guarantees. Furthermore, although it is also established that there is room for improvement in the scheduling, as shown in the previous chapter, replacing EASY Backfilling with

[2]The text of this chapter is adapted from the following published paper: **Danilo Carastan-Santos**, Raphael Y. de Camargo, Denis Trystram, Salah Zrigui. "*One can only gain by replacing EASY Backfilling: A simple scheduling policies case study.*". 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid), **2019**.

another algorithm might be seen as a risky change: one can see this change as a "jump into the dark", with the changes in performance only noticeable after a long period of time, and potentially after many strong-worded emails from many (important) users.

In this chapter we go towards bringing light to this jump. We selected a class of scheduling algorithms that keep the same simplicity and starvation guarantees of EASY Backfilling and we used a fast and reliable HPC simulation software to provide sound evidence on what could be gained – considering many relevant performance metrics – if one replaces EASY Backfilling. More specifically, in this chapter we present the following contributions:

- We present an experimental study that addresses the expectations and potential gains that come from replacing the EASY Backfilling scheduling policy in typical high-performance computing platforms;

- We highlight the Shortest Area First (SAF) scheduling policy, which, we argue, has the best-observed overall performance among the tested policies. In fact, we propose SAF as a new benchmark for future batch scheduling studies;

- We highlight an aspect that is often overlooked when evaluating the performance of a scheduling policy, which is the link between the number of resources used by jobs and the fairness of a given scheduling policy;

- We address the influence of the aggressive backfilling mechanism on the transparency and predictability of scheduling algorithms.

The remainder of this chapter is organized as follows. In Sections 4.1 and 4.2 we explain the scheduling problem under study and the performed experimental protocol. In Section 4.3 we present and discuss the obtained experimental results. Finally, we present a summary of the chapter in Section 4.4.

4.1 Preliminary Definitions

In this chapter we consider the same scheduling background presented in Section 3.2.1. We also use the logs (also called traces) present in the Parallel Workloads Archive, shared in the Standard Workload Format (SWF) [31]. We exploit the rich information present in these logs to drive the simulation workflow explained in Section 4.2. The main difference of this chapter in regards to Chapter 3 is that we consider three performance objectives, the waiting time (w_j, Equation 2.2), the bounded slowdown (bsld, Equation 3.2) and a third objective, called *per-processor bounded slowdown* [83] (pp-bsld or pp-slowdown, Equation 4.1), which is defined for a job j as:

$$\textit{pp-bsld}_j = \max\left(\frac{w_j + p_j}{q_j \cdot \max(p_j, \tau)}, 1\right) \tag{4.1}$$

where w_j is the waiting time of j and τ is a constant to prevent smaller jobs from reaching very high values, similarly with the bounded slowdown. The reasoning behind the per-processor bounded slowdown is to normalize the slowdown results for jobs who perform the same amount of work, though with different degrees of parallelism (number of processors). The pp-slowdown can be seen as a more appropriate objective for the parallel job scheduling problem, as it tries to balance the waiting time of the jobs in function of the number of processors q_j, which is not taken into account by either the waiting time or the slowdown.

Like other cost metrics, the waiting time, slowdown and pp-slowdown are usually considered in their cumulative versions, which means that one seeks to minimize the average waiting time, bsld or pp-bsld. It is worth noting that other metrics such as the maximum waiting time bsld or pp-bsld of all the jobs are also worthy of interest, though they must be taken more carefully, as we explain in Section 4.3.3.

For a queue of jobs Q, we consider the average waiting time, slowdown or pp-slowdown of Q as being the average of the respective metric, over all jobs $j \in Q$.

4.1.1 Fairness and User Satisfaction

Specifically for slowdown and pp-slowdown, the expectation of a good scheduling performance is that the waiting time of the jobs should be proportional of its running time, that is, a job that must perform a larger amount of work (and thus requires many resources and/or for a longer period) could "afford" a longer waiting time. Indeed, it is arguable that the slowdown metric can be a good performance metric for a job-centric fairness, in comparison to other metrics such as waiting time. One could envision, however, that a better performance metric could be a user-centric metric, that captures the overall satisfaction among users. Although this could be indeed the case, one can not simply simulate user behavior by reproducing a workload trace due to the fact that the workload would change (in an on-line manner) in function of the scheduler's performance (e.g. a more efficient scheduler would stimulate users to submit more jobs and vice versa). At the time of writing of this thesis, there is no consensus in the community about accurate and/or meaningful ways to simulate user behavior, which leads us to choose a job-centric approach rather than a user-centric one.

4.1.2 On-line Batch Scheduling Algorithm

In this chapter we consider a queue-ordering based, on-line parallel job scheduling algorithm that works as follows: the scheduler sorts – in increasing order according to a scheduling policy $f(j)$ – its waiting queue in two distinct events: (i) when a job arrives in the queue or (ii) when a resource (set of processors) is released and becomes available. When a job j is selected for execution and if the requested number of processors q_j is lower than the total number of processors available, then q_j processors are reserved for this job and they become unavailable. These processors will become available again only when p_j units of time have passed since the start of the execution of t. If the actual processing time p_j is larger than its estimate $\tilde{p}_j$, j is *killed*, that is, its execution is terminated.

Table 4.1 Scheduling policies used for comparison.

Name	Description	Function
FCFS	First-Come-First-Served [70]	$f(j) = r_j$
SPF	Smallest Estimated Processing Time First [71]	$f(j) = \tilde{p}_j$
SQF	Smallest Resource Requirement First	$f(j) = q_j$
SAF	Smallest Estimated "Area" First	$f(j) = \tilde{p}_j \cdot q_j$

In the case that there are not enough processors to process j, an *aggressive backfilling* subroutine [60] (see Section 2.1.2.3) is applied. In this case, it is estimated at which time there will be enough resources to process j. Next, the scheduler looks for jobs in the waiting queue – following the order of jobs already established by the scheduling policy $f(j)$ – for which there are enough processing resources and that do not delay the execution of j. If a job meets these aforementioned conditions, then it "jumps ahead" and is scheduled for execution.

A key component of this scheduling algorithm is the scheduling policy $f(j)$. Although many scheduling policies can be devised, in this work we are concerned in comparing *simple* scheduling policies. Table 4.1 shows the simple scheduling policies considered in this chapter. We define a scheduling policy $f(j)$ as *simple* if $f(j)$ is equal to one of the jobs' characteristics (notably FCFS, SPF and SQF, in which FCFS and SPF are well known policies in the off-line job scheduling literature) or its meaning is intuitive and transparent to the platform user (notably SAF, which sorts the jobs according to their "area" or "geometry"). One can observe that we could also envision the "last/largest" variant of the presented scheduling policies, namely the Last-Come-First-Served (LCFS), Largest Estimated Processing Time First (LPF), Largest Resource Requirement First (LQF) and Largest Area First (LAF) policies. However, we decided to not consider them because in our preliminary experiments, and as well as reported in the works of Gaussier *et al.* [39], the "last/largest" variants present consistent worse scheduling performances than their "first/shortest" variants.

4.1.2.1 Starvation Prevention

It is possible to observe that among all scheduling policies presented in Table 4.1, only FCFS can straightforwardly prevent starvation, that is, it guarantees that every job will eventually be processed. Therefore, some starvation prevention mechanism is mandatory for the remaining policies in order to be applicable in real scenarios. In this regard, we adopted a simple thresholding mechanism [39], in which a job j would receive a maximum priority (bypassing the priority given by the scheduling policy $f(j)$) if its waiting time exceeds a maximum threshold value Θ. If many jobs receive a maximum priority at the same time, they will follow a FCFS order.

The threshold value Θ is an important parameter and indeed must be set with some caution: a too small threshold value would constrain the scheduling policy $f(j)$, by assigning the maximum priority at too many jobs and thus too many jobs will follow FCFS order instead of $f(j)$. Conversely, a too large threshold value could be too prohibitive in the Quality of Service point of view of the platform. In this chapter we set the threshold value Θ as being three times the maximum processing time estimate allowed by the platform. In general, it is a common practice in HPC platforms to set a maximum processing time allowed, setting the threshold as being three times this value means that the slowdown of larger jobs (i.e. jobs with processing time close to the maximum) would have a value of around three. Since the policies used this chapter prioritize shorter/smaller jobs. This thresholding mechanism is dedicated to the longer/larger jobs and is proven to be effective in preventing starvation, as we show in Section 4.3 that the majority of jobs that reach the threshold are larger ones.

4.2 Experimental Workflow

In this section we present the experimental protocol adopted in this chapter. We make use of BatSim [21], which is a scientific instrument based on SimGrid [13] – a well known HPC systems simulator – specially tailored to simulate and study the behavior of batch scheduling algorithms. BatSim and SimGrid allows us to rapidly

Table 4.2 Real workload traces used for evaluation of the scheduling policies.

Name	Year	# CPUs	# Jobs	Util %	Duration
HPC2N	2002	240	202,871	60.1	42 Months
SDSC Blue	2000	1,152	243,306	76.7	32 Months
SDSC SP2	1998	128	59,715	83.4	24 Months
CTC SP2	1997	338	77,222	85.2	11 Months
KTH SP2	1996	100	28,476	70.1	11 Months

and accurately simulate the scheduling of many workload traces with using only a single workstation and in only a matter of days, which would not be feasible without simulation.

Table 4.2 shows the real workload traces used in this chapter. In order to provide statistically meaningful results with the scheduling of the traces, we adopted a sampling technique based on [26]. Algorithm 1 presents the pseudo-code. The idea is to generate new data using existing user profiles. A profile can be defined as the activity of a single user throughout the trace, split into many weekly time periods. To generate a new trace we combine several random permutations of each user's profiles. One can observe that this sampling technique is not capable of reflecting the workload changes in function of the scheduler's performance (as discussed in Section 4.1.1). However, it allows to generate as many logs as needed while preserving the jobs' properties of each user.

Algorithm 1 Workload trace resampling algorithm.

Require: list of user $profiles\ P$ extracted from the original workload trace. Number of weeks in the resampled trace n_w
Ensure: resampled trace W_{res}

$W_{res} \leftarrow \emptyset$
for $i = 1$ to n_w **do**
  $w_{res} \leftarrow \emptyset$
  for each user profile p in P **do**
    $p_{i_{res}} \leftarrow$ random weekly split from p
    add $p_{i_{res}}$ to w_{res}
  end for
  append w_{res} in W_{res}
end for
return W_{res}

For each trace of Table 4.2, we generate 10 samples using the aforementioned procedure. The size of each sample is proportional to the size of the original trace.

Each sample is then simulated following the scheduling algorithm presented in Section 4.1.2, taking into consideration each of the scheduling policies presented in Table 4.1. The results for each scheduling policy and workload trace presented in the next Section are statistical summaries of the ten samples of each trace.

4.3 Experimental Results

In this section we present the main results obtained by the experimental procedure described in Section 4.2. We perform several analysis in order to provide a better understanding of the behavior of the scheduling policies and what gains could be expected if a certain scheduling policy is chosen.

4.3.1 Overall Scheduling Performance

In this Section we aim to answer the following question: *What is the relative performance of the scheduling policies in Table 4.1?*

Figure 4.1 shows the overall performance results for the average slowdown, waiting time, and pp-slowdown. Each subplot refers to a workload trace from Table 4.2. To avoid outlier interference in the results, for each trace and scheduling policy we discarded the best performing and the worst performing workload sample (see Section 4.2) from the 10 initial workload samples. In other words, we present only the scheduling results of the samples whose performance belongs to the 10-90% percentile range. Each subplot contains statistics of the scheduling simulation of these remaining samples. The solid lines in the subplots represent the cumulative mean of the objective metric (average slowdown, waiting time, or pp-slowdown) of the finished jobs at each week of simulation, from the beginning to the end of the workload, and the dashed lines represent the cumulative maximum and minimum average values of the respective metric at each week.

Looking at the scheduling performance in Figure 4.1, we can cluster the tested policies in two classes: the ones that are oblivious of the processing time estimate $\tilde{p}$ (FCFS and SQF), and the ones that are not oblivious (SPF and SAF). From the

aforementioned Figure, we can observe a strong correlation between the scheduling performance of these clusters, with the former cluster consistently presenting worst performances than the latter. This result is expected: for the slowdown and pp-slowdown, jobs with a lower $\tilde{p}$ – and thus lower p, since $\tilde{p}$ is an upper bound of p – have a higher risk of inflating the metrics if they wait too much (see Equations 3.1 and 4.1). By favoring jobs with a lower $\tilde{p}$ (SPF and SAF), we assure that these high risk jobs are executed quickly, and thus the average for both slowdown and pp-slowdown are kept under control. The waiting time is also favored by prioritizing jobs with lower $\tilde{p}$, since for all traces these jobs are more frequent [24].

One point that is worth noticing is how much can be gained in quantitative values if a policy other than FCFS (notably SPF or SAF) is chosen and kept during a long period. In our experiments we achieve performance gains up to 83.4% (SPF), 61.4% (SAF), and 85.1% (SAF) for the slowdown, waiting time, and pp-slowdown respectively, in comparison with FCFS. It is important to note here that the scheduling simulation is performed with a starvation prevention mechanism (see Section 4.1.2). Therefore, these gains can be obtained while guaranteeing that no job will starve.

Another important observation is how SAF – which in contrast with SPF, is less known in the literature – performs consistently well in all objectives considered. We further address this phenomenon in the next section.

4.3.2 Is SAF the ultimate simple policy?

As highlighted in the previous section, the scheduling policies that are not oblivious to the processing time estimate $\tilde{p}$ (notably SPF and SAF) are the ones who achieved the most consistent good performances in the experiments that we performed. In this section we make a further analysis on which are the characteristics of the jobs that make them prioritized/delayed by these two policies, with an emphasis on the delayed jobs.

For the processing time estimate $\tilde{p}$ this analysis can be easily devised: SPF delays jobs with a larger $\tilde{p}$ and SAF is similar, with the distinction that it considers the number

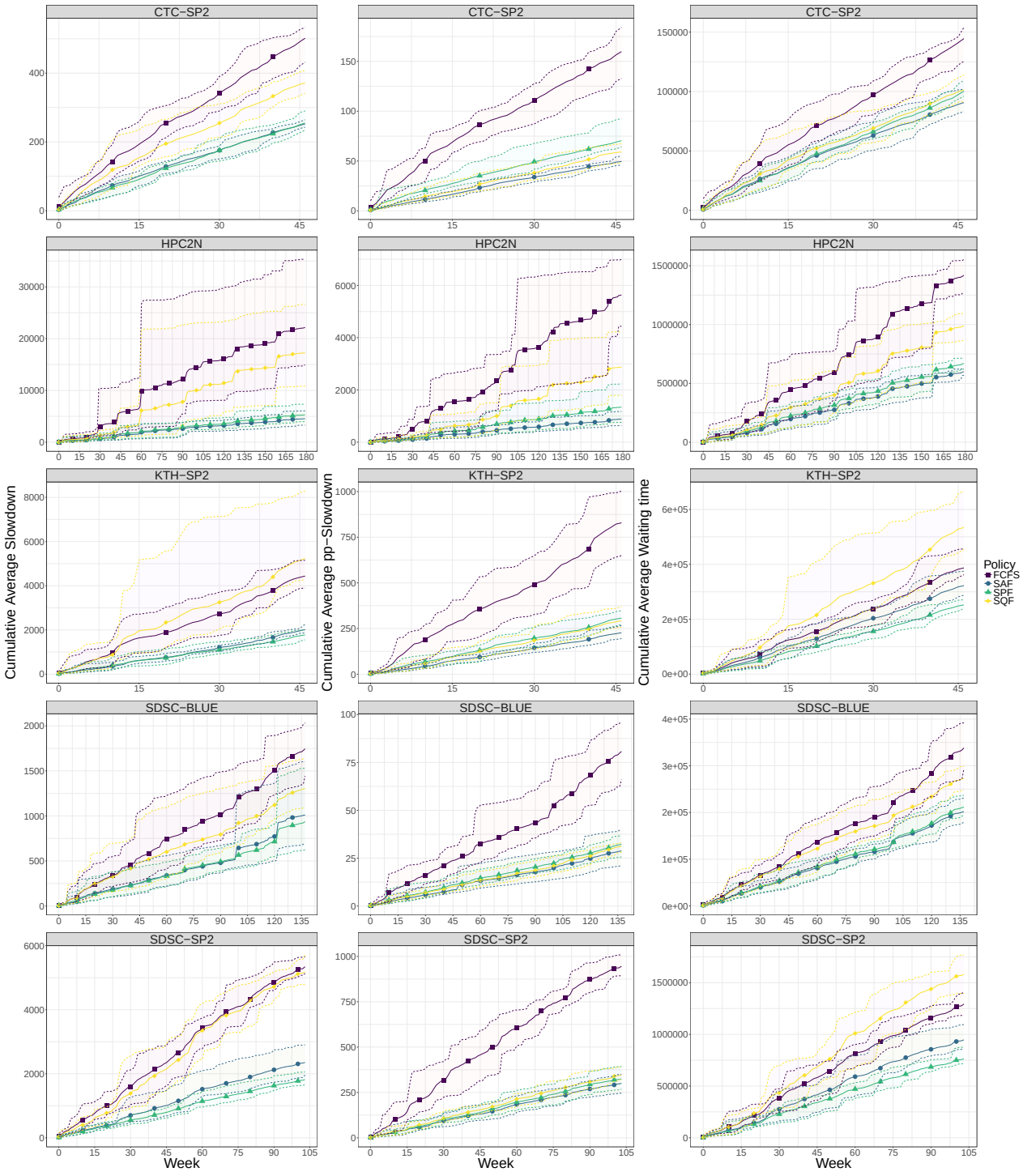


Figure 4.1 Cumulative weekly average slowdown, pp-slowdown and waiting time: For each trace, the middle solid line represents the mean and the two dashed lines represent the lower and upper 10-90 percentiles.

of processors q as well. This raises the importance of our thresholding mechanism, which specifically concerns jobs with a large $\tilde{p}$.

In its turn, for the number of processors q, Figure 4.2 shows the number of processors q of the top 100 jobs – of each sample of each trace – who got delayed the most (here defined as the jobs with the highest slowdown) for each scheduling policy. An interesting observation here is that SPF is oblivious to the number of processors q and thus no correlation should be expected for the delayed jobs in function of q. Therefore, SPF had a high risks of delaying jobs with smaller q which, in principle, should be easier to be scheduled in an HPC platform.

Indeed, we recall a known observation [29] that the slowdown and the waiting time metrics (arguably the most popular ones) do not take into consideration one important dimension of the scheduling problem: the number of requested processors q. Jobs that perform the same amount of work though with different shapes are treated indifferently by these metrics. The pp-slowdown generalizes the standard slowdown by including the number of processors q in the metric.

At this light SAF shows up as a solid policy among the simple ones we evaluated. It achieved close to best observed performances for the slowdown and waiting time objectives, and systematically outperformed all other simple policies for the pp-slowdown objective (Figure 4.1). This complies with the results obtained in the previous chapter (see Chapter 3), where the machine learned policies converged to functions that contain a SAF-like component. Although one can claim that SAF could be biased towards pp-slowdown, since with pp-slowdown we would seek to minimize an objective function that is related to the area of the jobs, we argue that the pp-slowdown is a more appropriate objective for the parallel job scheduling problem, in comparison with waiting time or slowdown.

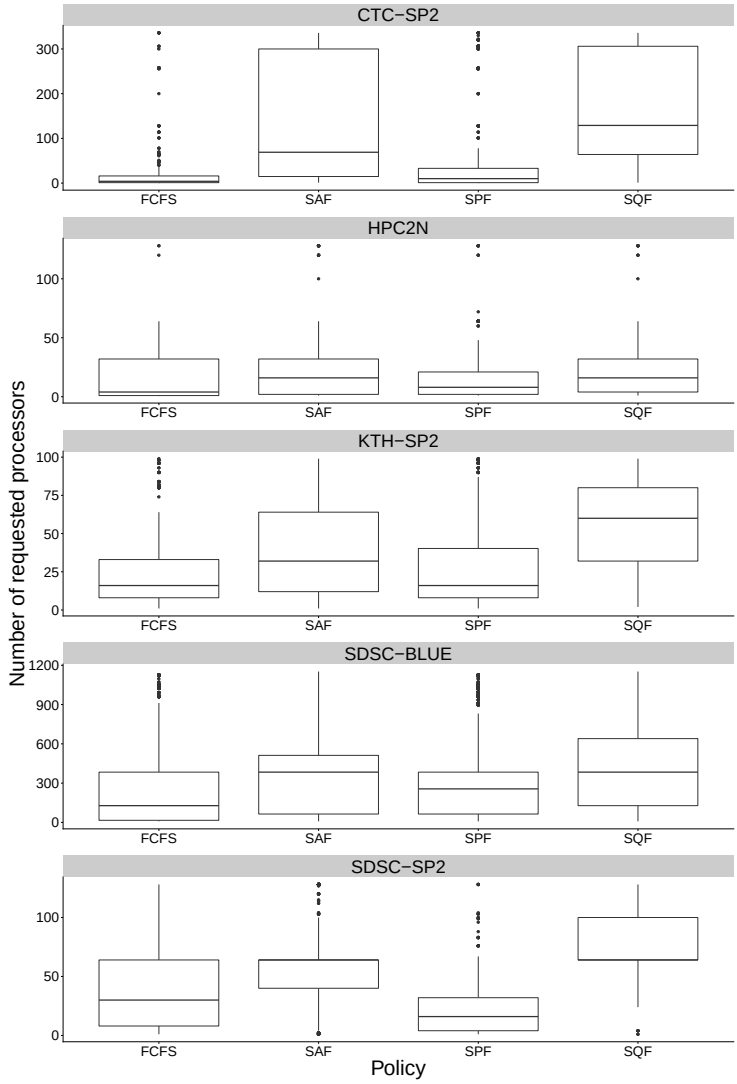


Figure 4.2 Number of processors of the top 100 jobs with highest slowdown values.

4.3.3 Accounting the Maximum: one should care with caution

One can notice that in this chapter we only seek to find good scheduling algorithms aiming at the average of the objective functions and not the maximum. Although one can argue that the maximum of the objective functions are important as well, in this section we present some observations found by our study that show that aiming only for the maximum can be potentially problematic.

The first point is that the maximum metric is centred at the performance of only one job, meaning that the value of the maximum can be unstable and subject to unpredictable factors, such as unavoidable bursts of jobs submissions and/or jobs that have some characteristic that can potentially mistakenly inflate the metric. To illustrate this potential, we clustered the jobs into two classes: the premature jobs, in which the difference between the processing time estimate $\tilde{p}$ and the actual processing time p is at least 100 times higher, and the standard jobs, which are the remaining jobs. Table 4.3 shows the percentage of premature jobs found for each workload trace. What is interesting to observe is that the number of premature jobs is not negligible, up to one third of all of the jobs of the trace. Furthermore, the difference between $\tilde{p}$ and p can be sometimes quite extreme: jobs that are marked as successful jobs (i.e. job that did not crash) and require the maximum processing time allowed $\tilde{p}$, though actually execute for around one minute happen in every trace. Since these jobs are marked as successful, we can not discard them from the analysis.

As a consequence, any scheduling policy that prioritizes jobs in function of the processing time estimate $\tilde{p}$ has a risk of delaying these premature jobs and, when evaluating the objective function of these jobs, they will obtain poor results which will harm the maximum of the objective function. To illustrate this effect, Table 4.4 shows the ratio between the average slowdown of the premature jobs and the average slowdown of the standard ones, for all traces and scheduling policies. We can notice that the difference in scheduling performance of these two classes of jobs

is large, up to 17 times larger for all policies in the HPC2N trace, and this difference in the maximum slowdown between these two classes (result not shown in Table 4.4) is even larger. We can also notice that this difference is often amplified by policies that takes $\tilde{p}$ into account (SPF and SAF).

Agreeing whether or not these performance gaps are due to the scheduler is always up to argument. However, Figure 4.3 shows a more holistic view of the scheduling performance: we grouped the jobs in many categories that are in function of the jobs' scheduling performance, from the jobs that were executed immediately (slowdown of 1), to the jobs that were poorly scheduled (slowdown of at least 100). We can observe that choosing another policy than FCFS shows performance improvements in all categories: the number of jobs who got executed immediately increases and the number of jobs in all other categories (the jobs who had to wait) decreases, with an exception of the SPF policy at the 1-10 slowdown range. These results are even more impressive for the category of jobs with poorer scheduling performances (100+ slowdown). For instance, by choosing SAF, the number of jobs who got badly scheduled can be lowered by more than half, up to 2.8x less poorly scheduled jobs in comparison with FCFS.

All of these points elucidate the importance of analyzing the scheduling performance in a holistic view, and the caution that must be taken into account when evaluating the scheduling performance with maximum values. We would certainly overlook these good properties of the studied scheduling policies if we had considered only the maximum of the objective functions.

Table 4.3 Percentage of premature jobs for each workload trace

Trace	% of premature jobs
HPC2N	17.4
SDSC Blue	30.2
SDSC-SP2	16.1
CTC-SP2	9.5
KTH-SP2	12.4

Figure 4.3 Distribution of the bounded slowdown values for all jobs

Table 4.4 Ratio of the average slowdown between the premature the standard jobs

Policy	HPC2N	SDSC Blue	SDSC SP2	CTC SP2	KTH SP2
FCFS	17.84	3.59	3.94	5.58	8.69
SPF	17.29	7.17	4.36	5.04	12.09
SQF	14.02	2.96	2.04	1.67	9.31
SAF	17.88	7.41	3.79	2.61	11.41

4.3.4 Backfilling Influence

One important question that raises when the queue ordering policy is changed (see Section 4.1.2) is how the backfilling mechanism behaves in function of the queue ordering policy. Although it is well known [60] that backfilling increases the platform's utilization and is unlikely to harm the original (without backfilling) schedule, its relevance in performance is not clear. This question is also worth of importance to bring a clearer notion about the predictability of the scheduling policies, that is, given one policy, how much it is likely that the jobs will actually follow such an order.

In order to clarify this point, for all samples of each trace and each scheduling policy we kept track on how many jobs got scheduled to execution by the backfilling mechanism. Figure 4.4 shows the distribution of the number of backfilled jobs over all samples, for each workload trace and scheduling policy. One interesting observation is the absence of backfilled jobs for the SQF policy for every trace and sample. This result is expected and we formalize it with the following proposition:

Proposition 1. *If the aggressive backfilling algorithm uses a queue of jobs sorted by SQF and there is no threshold mechanism added to the scheduling, no job is backfilled.*

Proof. As explained in Section 4.1.2, scheduling decisions are performed in two cases:

1. When a job arrives in the queue: in this case, let j_h be the job with the highest priority in the queue. Job j_h is in the queue, therefore there is not enough resources to process j_h. Since the queue is sorted by SQF order, there is no job in the queue that requires less resources than j_h, so none of them can

be backfilled. If a new job j arrives in the queue and its number of required processors is lower than the number of processors required by j_h, SQF will assign j with the highest priority and thus backfilling will no longer be applied for j. Conversely, where j requires more processors than j_h, j cannot be backfilled as aforementioned.

2. When a job is finished and its allocated resources are released: in this case, the jobs will be scheduled for execution following SQF order until it is no longer possible to schedule jobs with the current available resources. At this point, there are not enough resources to schedule the job with the highest priority in the queue and, since the queue is sorted in SQF order, no other job in the queue can be backfilled as aforementioned.

Since in both of the above cases it is impossible to backfill jobs, no jobs are backfilled.

□

Yet, some backfilling may happen when using SQF with jobs that exceeded the threshold in the waiting queue (since they break the SQF order). However, such jobs are expected to be very few. This explains some results found by Lelong *et al.* [53], in which they state that the SQF policy did not lead to many backfilling decisions in their experiments.

Interestingly, using SAF and SPF resulted in 78% and 56% less backfilled jobs on average, respectively, when compared to FCFS. Although it is unlikely that backfilling would harm the scheduling, as mentioned above, SPF and SAF are more consistent and predictable policies, since jobs are more likely to be scheduled for execution following the policy order, as oppose to being scheduled by "jumping ahead" in the waiting queue in unpredictable moments.

4.4 Summary

As the scale and power of high-performance computing (HPC) platforms increase, it becomes more crucial to deploy efficient resource management approaches (no-

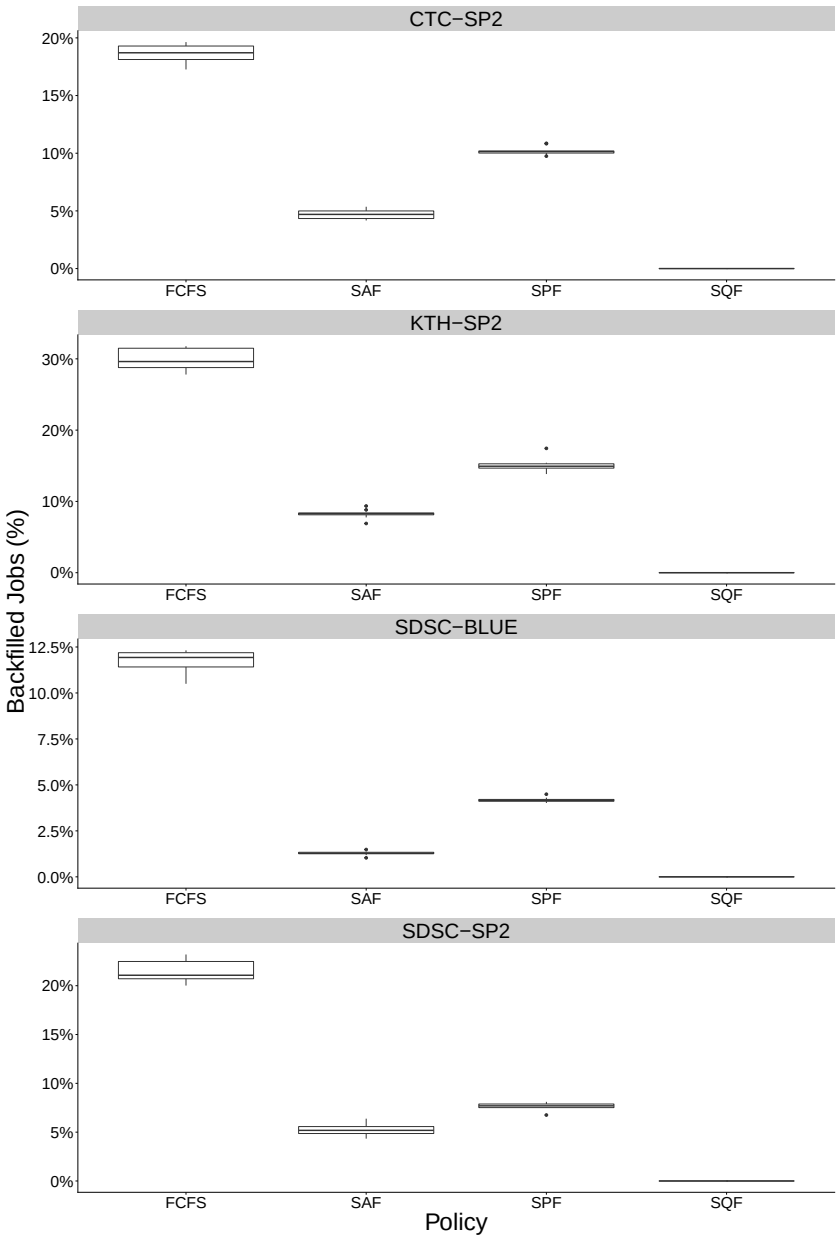


Figure 4.4 Distribution of backfilled jobs between resamplings.

tably scheduling algorithms) in order to prevent the dampening of such increase in computing power. In an adversarial manner, it is also important that such scheduling algorithms stay simple and easily understandable by the users. Furthermore, changing the scheduling algorithm is often seen as a risky move, mainly due to the possibility of having unseen and unpredictable changes in the performance of the platform, which could be detected only after a long period of time.

In this chapter, we move towards providing more knowledge and experience on what are the expectations if one decides to change the First-Come-First-Served (FCFS) scheduling policy with aggressive backfilling – the popular EASY Backfilling – scheduling algorithm. We selected a class of simple scheduling algorithms that differs from EASY Backfilling by changing the scheduling policy (other than FCFS) and adding a thresholding mechanism (to provide the same no starvation guarantees as FCFS). We used a flexible and reliable simulation software and exploited the rich information presented in HPC platform workload traces to find what could be observed and gained by using these other simple scheduling algorithms rather than EASY Backfilling.

Our results indicate that one can only gain by replacing EASY Backfilling with simple policies that consider the estimated processing time and the required resources, notably the Shortest Processing time First (SPF) and Shortest Area First (SAF). By adding a simple thresholding mechanism, it is possible to obtain significant performance improvements for the long run, using three relevant performance objectives, while also guaranteeing that every job in the waiting queue will eventually be executed. We show that these simple policies not only present better performance in average values, but they also significantly increase the number of jobs executed instantly (without waiting) and lower the number of jobs that wait for a long time. The performance gains over EASY Backfilling is distributed among all waiting jobs.

These simple policies also show that they can perform well with less interference from backfilling: the scheduler is more likely to follow the original order as set by the chosen scheduling policy, and not by the rules of backfilling, thus providing more

predictability and transparency, two properties that are sought by HPC platform administrators.

We also highlight a less known scheduling policy in the literature, the Shortest Area First (SAF). In our experimental campaign, we found that this policy managed to consistently provide close to the best (if not the best) observed performance in all scenarios and performance objectives we evaluated. For instance, considering the slowdown objective, SAF not only provided an average overall performance increase up to 83.4%, but as well increased the number of jobs that run immediately by up to 9% and lowered the number of jobs who waited for a long time (very long slowdown) by up to 2.8 times, in comparison with FCFS. This result reinforces the relevance of the jobs' area property, which was seen in the study performed in Chapter 3, and raises the question about possible analytical properties of SAF. Nevertheless, we reinforce that SAF must be considered as a baseline of comparison in future parallel job scheduling research.

Last but not least, we present some cautions that must be considered if one wants to provide a scheduling algorithm that minimizes the maximum of an objective function. Taking the slowdown objective function as an example, we observed a class of jobs whose presence in the workload is not negligible and can mistakenly lead to inflated maximum slowdown values. If one only looks at the maximum of an objective function to evaluate the scheduling performance, some good scheduling policies (as the aforementioned ones) can be overlooked.

There is still work to be done in this subject: the first point is to address how users play a role in the global scheduling. As mentioned in Section 4.1.1, our approach on evaluating the scheduling performance is centred at the platform and the jobs, where users are not taken into account. An ideal scenario would be to simulate the users reacting to the scheduler's performance. In this regard, accurate and reliable user models are required to properly simulate user behavior. Another point is how we can exploit SAF to provide SAF-like scheduling policies that are adapted to certain situations and/or time periods.

5 General Discussion

5.1 General Remarks and Future Research Directions

In this thesis we aimed to increase the computing efficiency of High Performance Computing (HPC) platforms by performing a better management of its resources. Informally, an HPC platform would be more efficient if it is able to process applications faster. Naively this could be solved by increasing the speed of the processors or adding more processors to the platform. However, as discussed in Chapter 1, making a better resource usage of the current HPC platforms is a matter of great importance, since the power consumption of such HPC platforms is arguably too high already.

A better usage of an HPC platform can be achieved by making better decisions as to when and where the applications (jobs) will be executed in the platform: the so called job scheduling. Job scheduling in HPC platforms, however, is a notoriously complex and delicate problem. Many parallel job scheduling problems (see Chapter 2) fall into the NP-Hard class of problems. As a consequence, many scheduling algorithms with strong performance guarantees are too convoluted and computationally expensive, thus hindering their deployment in practical situations. The majority of HPC platform maintainers choose instead simple list scheduling algorithms based into a backfilling mechanism, with arguably the simplest queue ordering policy: the fist job that arrives is the first to be processed, the so called First-Come-First-Served (FCFS).

At a first moment (Chapter 3) we explored the jobs' and the HPC platform characteristics in an attempt to devise simple scheduling heuristics for parallel job scheduling. The main hypothesis was that we could obtain novel scheduling patterns by analyzing and making use of HPC workload execution logs. For this task we used

automatic, machine learning (ML) procedures to find these patterns, which were encapsulated by simple nonlinear functions of the jobs' characteristics. We presented experimental evidence that the patterns encapsulated by these functions can in fact result in better scheduling performance if these functions are used as a scheduling policy.

As advocated in Section 2.2, finding relevant patterns in data is rather dependant on pioneer insights than making usage of machine learning methods themselves. In this regard, one of the pioneer insights of Chapter 3 was about how we use the workload data to drive ML algorithms. In Section 3.2.2 we proposed a method to generate derivative data based on the actual workload logs, which measures how much is the impact on the scheduling performance if a certain job type is selected for processing. This derivative data, in conjunction with the jobs' characteristics, were afterwards used to train ML models. This derivative data was obtained by performing fast scheduling simulations – which were only possible thanks to fast and reliable HPC simulation software such as SimGrid [13] – taking into account the jobs' data present in the workload logs.

Beyond pioneer insights, finding relevant patterns in data can also rely on good design choices. Taking the non-linear function form proposed in Section 3.2.3 as candidate regression models was a fortunate choice that not only resulted in good scheduling policies, but as well highlighted the importance of the "area" feature of the jobs, which was one of the motivations behind the work performed in Chapter 4. These results, however, do not disqualify at all other functional forms such as polynomials of the jobs' characteristics. These other functional forms may have the same potential to provide efficient scheduling policies, though perhaps not in the same manner that we achieved with the nonlinear functional form we used. For instance, Legrand *et al.* [52] has shown afterwards that – by using a black box optimization algorithm to fit the models – simple linear combinations of jobs characteristics can perform well as scheduling policies. Addressing whether polynomials are good functional forms for scheduling policies is an equally interesting future research direction.

Another interesting observation is the hindrance in finding models that are both efficient and transparent. This effect is aggravated in the task of finding models to be used as scheduling policies, because the models efficiency will be evaluated by the scheduling performance of the learned policies, and not just by the prediction accuracy of the target values. In Chapter 3 we opted to choose simple nonlinear functional forms in an attempt to bring both simplicity and nonlinear flexibility for the models. Nevertheless, it is still hard to reason over some aspects of the learned policies. For instance, it is nontrivial to reason over the "magic" values present in the coefficients, as well as it is nontrivial to reason over the emergence of square roots and logarithms in the functions F1 to F4 (see Table 3.2). This lack of full semantic meaning of the learned models is arguably a known problem in machine learning and it must e taken into account when one seeks to learn scheduling policies. Finding scheduling models through ML that perform well and have clear reasoning behind its decisions is a noticeable challenge.

However, from the functions F1 to F4 we can observe the prominence of the "area" feature of the jobs, which lead to the hypothesis that the Shortest Area First (SAF) can be an efficient scheduling policy. Before the work performed in Chapter 4, SAF was a known policy though rarely used in scheduling performance evaluation studies. In Chapter 4 we show experimental evidence that SAF can provide noticeable scheduling performance improvements (specially in regards to FCFS), while keeping an equivalent level of simplicity and transparency in comparison to FCFS: SAF is just the processing time estimate multiplied by the number of processors, there is no "magic" values or unexplained square roots or logarithms.

In fact, a later experiment (results not shown in this thesis) show that functions F1 to F4 only outperform SAF by a small margin, and in some cases their scheduling performances are equivalent. The same phenomenon happened with the linear combinations proposed in the work of Legrand *et al.* [52]. At the light of these observations, an open question is how (or can we) outperform SAF by a large margin, while keeping a similar level of simplicity of a queue ordering, list scheduling policy

constituted by simple functions such as linear functions, nonlinear arbitrary functions, or polynomials.

In regards to the scheduling performance, we can perhaps conceive better performances by utilizing more complex decision making models. One can use, for instance, a reinforcement learning [74] approach, where the scheduler is modeled as an agent that learns good scheduling strategies "on the fly" in function of the consequences (rewards) of its scheduling decisions (actions). There exist many advanced reinforcement learning algorithms that are being noticeably successful in other decision making processes such as artificial intelligence (AI) models for games such as Chess and GO [69]. Although we envision these approaches as likely candidates to provide good scheduling performances, the trade-off between performance and transparency of the models will sill be present. At last, another interesting research direction is to perform a careful data analysis on the data generated by the simulation procedure presented in Section 3.2.2. This data analysis has potential to provide simple intuitions about good scheduling strategies.

At a second moment (Chapter 4), rather than devising novel scheduling policies for parallel job scheduling, we explored a small and known set of scheduling policies (Table 4.1) and we sought to provide insights on what we can expect if we replace the FCFS scheduling policy of EASY Backfilling with one of the policies from this small set. This study was mainly motivated by the fact that FCFS is arguably the simplest scheduling policy available, and brings convenient guarantees such as no-starvation. Adding the aggressive backfilling into the algorithm increases the overall resource utilization and it is being largely adopted by practitioners, therefore the performance of EASY Backfilling is well known, providing a "comfort zone" for practitioners. Replacing FCFS with any other policy can be seen as high-risk change, mainly because the performance effects of another policy will only be noticeable in the long run, and many practitioners simply cannot afford this risk.

For the scheduling policies we selected policies that are simple combinations of the jobs' characteristics, notably the estimated processing time ($\tilde{p}$), the requested number of processors (q), and the release date (r). We also added a simple waiting

time threshold mechanism to provide the same no-starvation guarantees of FCFS for these other policies. Using simulations we performed a holistic, careful experimental campaign to see what are the long term scheduling performances of the policies under different jobs and HPC platforms characteristics. Based on the experimental results, a first observation is the prominence of two scheduling policies, the Shortest estimated Processing time First (SPF) and SAF, with consistent better scheduling performances in comparison with FCFS. We show further experimental evidence that goes towards confirming the hypothesis of the good efficiency of SAF, raised in Chapter 3. A second observation is in regards to the backfilling efficiency: we observed that in all cases where backfilling is possible, backfilling helps in increasing performance, though it does not outperform a more efficient sorting of the waiting queue, such as the sorting performed by SPF and SAF. The work performed has shed some light at moving towards replacing EASY Backfilling as the default choice of scheduling policy.

Yet, many assumptions had to be taken in order to perform the study presented at this thesis, each of which arguably drives the work away from real scenarios. The first noticeable assumption is the homogeneous HPC platform. A more realistic scenario would be an heterogeneous HPC platform constituted by either CPUs with different speeds or a CPU/GPU platform, where the nodes have accelerator devices (such as GPUs) in addition to CPUs to accelerate the jobs processing (if the jobs are implemented to support accelerator devices). Such study would require workload logs with information such as the jobs allocation and computing nodes characteristics, which are quite rare in HPC workload logs. Furthermore, a more detailed scheduling simulation would take more time to process, specially when simulating longer periods such as months. Faster simulation algorithms may be required in such cases. On the subject of HPC platform configurations, another point is that we do not take into account the memory hierarchical organization of the HPC platforms, since taking this aspect into account has the same aforementioned practical hindrances.

Another assumption is that the users do not affect the scheduling performance, which is definitely not true in real scenarios. As discussed many times during this manuscript, users may for instance submit more or less jobs, in the case of a good or poor scheduling performance, respectively, which is already a sign of user interference. A more realistic scenario would be for instance to create job workloads taking into account the user awareness into the scheduling. Such task may be achievable for instance by game theory models, where users are players who compete between themselves and the scheduler to process the highest number of jobs as possible, or process their jobs as soon as possible. Although conceiving such models is an achievable task, validating such models is nontrivial and requires further research.

Other research directions that can drive the work closer to real scenarios is to consider other scheduling constraints such as moldable jobs rather than rigid jobs. As discussed in Chapter 2, moldable jobs are often the default scenario for Cloud Computing or Big Data workloads, which are quite common real-world situations. Knowing how the level of parallelism affects the jobs processing time, performing the same study accomplished in this thesis taking into account moldable jobs is an achievable task. At last, other constraints may be taken into account such as job preemptions, job processing reservations – where the users can reserve at which point in time they want their jobs executed – and flexible processing time estimates, where users can extend their jobs' processing time estimate during their processing.

5.2 Work Dissemination

Many communications arose from the work performed during this thesis. The following first set presents the communications that are directly related to the contributions presented in this thesis:

- **Danilo Carastan-Santos**, and Raphael Y. de Camargo. "*Obtaining dynamic scheduling policies with simulation and machine learning*". In the International Conference for High Performance Computing, Networking, Storage and Analy-

sis (SC), ACM Press, **2017**. Full paper at the main track of one of the major international conferences in High Performance Computing. This paper comprises the contributions presented in Chapter 3 and was also nominated for both Best Paper and Best Student Paper awards of the conference.

- **Danilo Carastan-Santos**, Raphael Y. de Camargo, Denis Trystram, Salah Zrigui. "*One can only gain by replacing EASY Backfilling: A simple scheduling policies case study.*". 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid), **2019**. Full paper at the main track of a well recognized international conference in High Performance Computing. This paper comrpises the contributions presented in Chapter 4 and also was the Best Paper Award winner of the conference at the aforementioned year.

The following second set presents the communications that arose in a satellite manner – in the form of collaborations and/or supplementary work – during the thesis. All of these works relate to the thesis subject in some way, either by the HPC resource management or the HPC applications aspects:

- Luis Sant'Ana, **Danilo Carastan-Santos**, Daniel Cordeiro and Raphael Y. de Camargo. "*Real-Time Scheduling Policy Selection from Queue and Machine States.*". 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid), **2019**. Full paper at the main track of a well recognized international conference in High Performance Computing.

- **Danilo Carastan-Santos**, David C. Martins-Jr, Siang W. Song, Luiz C. S. Rozante, and Raphael Y. de Camargo. "*A hybrid CPU-GPU-MIC algorithm for minimal hitting set enumeration.*". Concurrency and Computation: Practice and Experience, **2018**. Full paper in well recognized journal in High Performance Computing.

- Luis Sant'Ana, **Danilo Carastan-Santos**, Daniel Cordeiro and Raphael Y. de Camargo. "*Analysis of Potential Online Scheduling Improvements by Real-Time Strategy Selection.*". XIX Simpósio de Computação de Alto-Desempenho (WS-

CAD), **2018**. Full paper in a recognized Brazilian High Performance Computing workshop.

- **Danilo Carastan-Santos**, David C. Martins-Jr, Siang W. Song, Luiz C. S. Rozante, and Raphael Y. de Camargo. "*A hybrid CPU-GPU-MIC algorithm for hitting set problem.*". XVIII Simpósio de Computação de Alto-Desempenho (WSCAD), **2017**. Full paper in a recognized Brazilian High Performance Computing workshop.

Bibliography

[1]Abdel Krim Amoura, Evripidis Bampis, Claire Kenyon, and Yannis Manoussakis. „Scheduling independent multiprocessor tasks“. In: *Algorithmica* 32.2 (2002), pp. 247–261 (cit. on p. 17).

[2]Brenda S Baker, Edward G Coffman Jr, and Ronald L Rivest. „Orthogonal packings in two dimensions“. In: *SIAM Journal on computing* 9.4 (1980), pp. 846–855 (cit. on p. 17).

[3]Michael A Bender, Soumen Chakrabarti, and Sambavi Muthukrishnan. „Flow and Stretch Metrics for Scheduling Continuous Job Streams.“ In: *SODA*. Vol. 98. 1998, pp. 270–279 (cit. on p. 17).

[4]Evan Berkowitz, M. A. Clark, Arjun Gambhir, et al. „Simulating the Weak Death of the Neutron in a Femtoscale Universe with Near-exascale Computing“. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*. SC '18. Dallas, Texas: IEEE Press, 2018, 55:1–55:9 (cit. on p. 1).

[5]Michael R Berthold, Christian Borgelt, Frank Höppner, and Frank Klawonn. *Guide to intelligent data analysis: how to intelligently make sense of real data*. Springer Science & Business Media, 2010 (cit. on pp. 27, 29).

[6]Abhinav Bhatele, Kathryn Mohror, Steve H. Langer, and Katherine E. Isaacs. „There Goes the Neighborhood: Performance Degradation due to Nearby Jobs“. In: *SC*. ACM, Nov. 2013, 41:1–41:12 (cit. on p. 15).

[7]Christopher M Bishop. *Pattern recognition and machine learning*. springer, 2006 (cit. on pp. 27, 29, 30).

[8]Raphaël Bleuse. „Apprehending heterogeneity at (very) large scale. (Appréhender l'hétérogénéité à (très) grande échelle)“. PhD thesis. Grenoble Alpes University, France, 2017 (cit. on pp. 16, 17).

[9]Marin Bougeret, Pierre-François Dutot, Klaus Jansen, Christina Otte, and Denis Trystram. „Approximation Algorithms for Multiple Strip Packing“. In: *Approximation and Online Algorithms, 7th International Workshop, WAOA 2009, Copenhagen, Denmark, September 10-11, 2009. Revised Papers*. 2009, pp. 37–48 (cit. on p. 17).

[10]Peter Brucker. *Scheduling Algorithms*. Fifth Edition. Springer, 2007 (cit. on pp. 2, 6, 8, 13).

[11]Nicolas Capit, Georges Da Costa, Yiannis Georgiou, et al. „A batch scheduler with high level components“. In: *Cluster Computing and the Grid, 2005. CCGrid 2005. IEEE International Symposium on*. Vol. 2. IEEE. 2005, pp. 776–783 (cit. on p. 22).

[12]Raymond J Carroll and David Ruppert. *Transformation and weighting in regression*. Vol. 30. CRC Press, 1988 (cit. on p. 40).

[13]Henri Casanova, Arnaud Giersch, Arnaud Legrand, Martin Quinson, and Frédéric Suter. „Versatile, Scalable, and Accurate Simulation of Distributed Applications and Platforms“. In: *Journal of Parallel and Distributed Computing* 74.10 (June 2014), pp. 2899–2917 (cit. on pp. 41, 42, 64, 80).

[14]Walfredo Cirne and Francine Berman. „A model for moldable supercomputer jobs“. In: *Proceedings 15th International Parallel and Distributed Processing Symposium. IPDPS 2001*. IEEE. 2000, 8–pp (cit. on p. 15).

[15]Adaptative Computing. *Moab Workload Manager Documentation*. `http://www.adaptivecomputing.com/support/documentation-index/`. Accessed: 2017-03-26. 2017 (cit. on p. 22).

[16]Víctor Cuevas-Vicenttín, Saumen Dey, Sven Köhler, Sean Riddle, and Bertram Ludäscher. „Scientific workflows and provenance: Introduction and research opportunities“. In: *Datenbank-Spektrum* 12.3 (2012), pp. 193–203 (cit. on p. 14).

[17]Leonardo Dagum and Ramesh Menon. „OpenMP: An Industry-Standard API for Shared-Memory Programming“. In: *IEEE Comput. Sci. Eng.* 5.1 (Jan. 1998), pp. 46–55 (cit. on p. 13).

[18]Charles Darwin. *On the origin of species, 1859*. Routledge, 2004 (cit. on p. 25).

[19]Jianzhong Du and Joseph Y-T Leung. „Complexity of scheduling parallel task systems“. In: *SIAM Journal on Discrete Mathematics* 2.4 (1989), pp. 473–487 (cit. on p. 17).

[20]Alejandro Duran and Michael Klemm. „The Intel® many integrated core architecture“. In: *High Performance Computing and Simulation (HPCS), 2012 International Conference on*. IEEE. 2012, pp. 365–366 (cit. on pp. 14, 58).

[21]Pierre-François Dutot, Michael Mercier, Millian Poquet, and Olivier Richard. „Batsim: A Realistic Language-Independent Resources and Jobs Management Systems Simulator“. In: *Job Scheduling Strategies for Parallel Processing*. Ed. by Narayan Desai and Walfredo Cirne. Cham: Springer International Publishing, 2017, pp. 178–197 (cit. on p. 64).

[22]Pierre-François Dutot, Erik Saule, Abhinav Srivastav, and Denis Trystram. „Online Non-preemptive Scheduling to Optimize Max Stretch on a Single Machine“. In: *Computing and Combinatorics - 22nd International Conference, COCOON 2016, Ho Chi Minh City, Vietnam, August 2-4, 2016, Proceedings*. 2016, pp. 483–495 (cit. on p. 17).

[23]Umer Farooq, Zied Marrakchi, and Habib Mehrez. „FPGA Architectures: An Overview“. In: *Tree-based Heterogeneous FPGA Architectures: Application Specific Exploration and Optimization*. New York, NY: Springer New York, 2012, pp. 7–48 (cit. on p. 14).

[24]Dror Feitelson. *Parallel Workloads Archive: Logs*. `http://www.cs.huji.ac.il/labs/parallel/workload/logs.html`. Online; last access 30 november 2018. 2018 (cit. on p. 67).

[25]Dror G Feitelson. „Metrics for parallel job scheduling and their convergence“. In: *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer. 2001, pp. 188–205 (cit. on p. 36).

[26]Dror G Feitelson. „Resampling with feedback — a new paradigm of using workload data for performance evaluation“. In: *European Conference on Parallel Processing*. Springer. 2016, pp. 3–21 (cit. on pp. 12, 65).

[27]Dror G Feitelson. *Workload modeling for computer systems performance evaluation*. Cambridge University Press, 2015 (cit. on p. 25).

[28]Dror G Feitelson and Morris A Jettee. „Improved utilization and responsiveness with gang scheduling“. In: *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer. 1997, pp. 238–261 (cit. on p. 23).

[29]Dror G Feitelson and Larry Rudolph. „Metrics and benchmarking for parallel job scheduling“. In: *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer. 1998, pp. 1–24 (cit. on p. 69).

[30]Dror G Feitelson, Larry Rudolph, Uwe Schwiegelshohn, Kenneth C Sevcik, and Parkson Wong. „Theory and practice in parallel job scheduling“. In: *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer. 1997, pp. 1–34 (cit. on pp. 15, 21).

[31]Dror G. Feitelson, Dan Tsafrir, and David Krakov. „Experience with using the Parallel Workloads Archive“. In: *Journal of Parallel and Distributed Computing* 74.10 (2014), pp. 2967–2982 (cit. on p. 61).

[32]Dror G Feitelson, Dan Tsafrir, and David Krakov. „Experience with using the parallel workloads archive“. In: *Journal of Parallel and Distributed Computing* 74.10 (2014), pp. 2967–2982 (cit. on pp. 36, 50).

[33]Message P Forum. *MPI: A Message-Passing Interface Standard*. Tech. rep. Knoxville, TN, USA: University of Tennessee Knoxville, 1994 (cit. on p. 13).

[34]Eitan Frachtenberg and Dror G Feitelson. „Pitfalls in parallel job scheduling evaluation“. In: *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer. 2005, pp. 257–282 (cit. on p. 12).

[35]Jerome Friedman, Trevor Hastie, and Robert Tibshirani. *The elements of statistical learning*. Vol. 1. Springer series in statistics New York, 2001 (cit. on p. 27).

[36]Haohuan Fu, Conghui He, Bingwei Chen, et al. „18.9Pflopss Nonlinear Earthquake Simulation on Sunway TaihuLight: Enabling Depiction of 18-Hz and 8-meter Scenarios“. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '17. Denver, Colorado: ACM, 2017, 2:1–2:12 (cit. on p. 1).

[37]Ana Gainaru, Guillaume Aupy, Anne Benoit, et al. „Scheduling the I/O of HPC Applications Under Congestion“. In: *IPDPS*. IEEE, May 2015, pp. 1013–1022 (cit. on p. 17).

[38]Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, Jan. 1979 (cit. on p. 13).

[39]E. Gaussier, J. Lelong, V. Reis, and D. Trystram. „Online Tuning of EASY-Backfilling using Queue Reordering Policies“. In: *IEEE Transactions on Parallel and Distributed Systems* 29.10 (Oct. 2018), pp. 2304–2316 (cit. on pp. 3, 19, 63, 64).

[40]Eric GAUSSIER, David GLESSER, Valentin REIS, and Denis TRYSTRAM. „Improving Backfilling by Using Machine Learning to Predict Running Times“. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '15. Austin, Texas: ACM, 2015, 64:1–64:10 (cit. on pp. 3, 25, 56).

[41]Y GEORGIOU. „Resource and Job Management in High Performance Computing“. PhD thesis. PhD Thesis, Joseph Fourier University, France, 2010 (cit. on p. 22).

[42]Ian GOODFELLOW, Yoshua BENGIO, and Aaron COURVILLE. *Deep learning*. MIT press, 2016 (cit. on pp. 27, 30).

[43]Ronald Lewis GRAHAM, Eugene Leighton LAWLER, Jan Karel LENSTRA, and Alexander Hendrik George RINNOOY KAN. „Optimization and Approximation in Deterministic Sequencing and Scheduling: a Survey“. In: *Annals of Discrete Mathematics* 5.2 (1979), pp. 287–326 (cit. on p. 6).

[44]Johann L HURINK and Jacob Jan PAULUS. „Online algorithm for parallel job scheduling and strip packing“. In: *International Workshop on Approximation and Online Algorithms*. Springer. 2007, pp. 67–74 (cit. on p. 17).

[45]Klaus JANSEN. „A (3/2+ ε) approximation algorithm for scheduling moldable and non-moldable parallel tasks“. In: *Proceedings of the twenty-fourth annual ACM symposium on Parallelism in algorithms and architectures*. ACM. 2012, pp. 224–235 (cit. on p. 17).

[46]Klaus JANSEN and Lorant PORKOLAB. „Linear-time approximation schemes for scheduling malleable parallel tasks“. In: *Algorithmica* 32.3 (2002), pp. 507–520 (cit. on p. 17).

[47]Eric JONES, Travis OLIPHANT, Pearu PETERSON, et al. *SciPy: Open source scientific tools for Python*. 2001– (cit. on p. 42).

[48]Hans KELLERER, Thomas TAUTENHAHN, and Gerhard WOEGINGER. „Approximability and nonapproximability results for minimizing total flow time on a single machine“. In: *SIAM Journal on Computing* 28.4 (1999), pp. 1155–1166 (cit. on p. 17).

[49]Johann KEPLER. „Astronomia nova.“ In: *(Pragae) 1609* (2015) (cit. on p. 25).

[50]CHUNG-YEE LEE and XIAOQIANG CAI. „Scheduling one and two-processor tasks on two parallel processors“. In: *IIE transactions* 31.5 (1999), pp. 445–455 (cit. on p. 17).

[51]Arnaud LEGRAND, Alan SU, and Frédéric VIVIEN. „Minimizing the stretch when scheduling flows of divisible requests“. In: *Journal of Scheduling* 11.5 (2008), pp. 381–404 (cit. on p. 17).

[52]Arnaud LEGRAND, Denis TRYSTRAM, and Salah ZRIGUI. „Adapting Batch Scheduling to Workload Characteristics: What can we expect From Online Learning?“ In: *34th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 2019 (cit. on pp. 3, 80, 81).

[53]Jérôme LELONG, Valentin REIS, and Denis TRYSTRAM. „Tuning EASY-Backfilling Queues“. In: *21st Workshop on Job Scheduling Strategies for Parallel Processing*. 2017 (cit. on pp. 25, 38, 75).

[54]David A LIFKA. „The anl/ibm sp scheduling system“. In: *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer. 1995, pp. 295–303 (cit. on p. 24).

[55]Uri LUBLIN and Dror G FEITELSON. „The workload on parallel supercomputers: modeling the characteristics of rigid jobs“. In: *Journal of Parallel and Distributed Computing* 63.11 (2003), pp. 1105–1122 (cit. on pp. 37, 44, 46, 48, 55, 56).

[56]Giorgio Lucarelli, Fernando Machado Mendonça, Denis Trystram, and Frédéric Wagner. „Contiguity and Locality in Backfilling Scheduling“. In: *CCGRID*. IEEE, May 2015, pp. 586–595 (cit. on p. 20).

[57]Giorgio Lucarelli, Benjamin Moseley, Nguyen Kim Thang, Abhinav Srivastav, and Denis Trystram. „Online Non-Preemptive Scheduling to Minimize Weighted Flow-time on Unrelated Machines“. In: *CoRR* abs/1804.08317 (2018). arXiv: 1804.08317 (cit. on p. 17).

[58]Scott M Lundberg and Su-In Lee. „A Unified Approach to Interpreting Model Predictions“. In: *Advances in Neural Information Processing Systems 30*. Ed. by I. Guyon, U. V. Luxburg, S. Bengio, et al. Curran Associates, Inc., 2017, pp. 4765–4774 (cit. on p. 30).

[59]Cathy McCann, Raj Vaswani, and John Zahorjan. „A dynamic processor allocation policy for multiprogrammed shared-memory multiprocessors“. In: *ACM Transactions on Computer Systems (TOCS)* 11.2 (1993), pp. 146–178 (cit. on p. 23).

[60]Ahuva W. Mu'alem and Dror G. Feitelson. „Utilization, predictability, workloads, and user runtime estimates in scheduling the IBM SP2 with backfilling“. In: *IEEE Transactions on Parallel and Distributed Systems* 12.6 (2001), pp. 529–543 (cit. on pp. 2, 19, 21–24, 59, 63, 74).

[61]Bill Nitzberg, Jennifer M Schopf, and James Patton Jones. „PBS Pro: Grid computing and scheduling attributes“. In: *Grid resource management*. Springer, 2004, pp. 183–190 (cit. on pp. 2, 22).

[62]J. D. Owens, M. Houston, D. Luebke, et al. „GPU computing“. In: *Proceedings of the IEEE* 96.5 (2008), pp. 879–899 (cit. on pp. 1, 14, 30, 58).

[63]Dejan Perkovic and Peter J Keleher. „Randomization, speculation, and adaptation in batch schedulers“. In: *Proceedings of the 2000 ACM/IEEE conference on Supercomputing*. IEEE Computer Society. 2000, p. 7 (cit. on p. 25).

[64]Michael L. Pinedo. *Scheduling: Theory, Algorithms, and Systems*. 3rd. Springer Publishing Company, Incorporated, 2008 (cit. on pp. 2, 6, 10, 13).

[65]Millian Poquet. „Simulation approach for resource management. (Approche par la simulation pour la gestion de ressources)“. PhD thesis. Grenoble Alpes University, France, 2017 (cit. on p. 21).

[66]Gonzalo P Rodrigo, P-O Östberg, Erik Elmroth, et al. „Towards understanding HPC users and systems: a NERSC case study“. In: *Journal of Parallel and Distributed Computing* 111 (2018), pp. 206–221 (cit. on p. 22).

[67]L. F. Sant'Ana, D. Carastan-Santos, D. Cordeiro, and R. de Camargo. „Analysis of Potential Online Scheduling Improvements by Real-Time Strategy Selection“. In: *2018 Symposium on High Performance Computing Systems (WSCAD)*. Oct. 2018, pp. 1–7 (cit. on p. 38).

[68]George AF Seber and Christopher John Wild. „Nonlinear Regression. Hoboken“. In: *New Jersey: John Wiley & Sons* 62 (2003), p. 63 (cit. on p. 30).

[69]David Silver, Thomas Hubert, Julian Schrittwieser, et al. „A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play“. In: *Science* 362.6419 (2018), pp. 1140–1144 (cit. on pp. 1, 82).

[70] Joseph SKOVIRA, Waiman CHAN, Honbo ZHOU, and David LIFKA. „The EASY—LoadLeveler API Project“. In: *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer. 1996, pp. 41–47 (cit. on pp. 42, 63).

[71] Srividya SRINIVASAN, Rajkumar KETTIMUTHU, Vijay SUBRAMANI, and P SADAYAPPAN. „Characterization of backfilling strategies for parallel job scheduling“. In: *Parallel Processing Workshops, 2002. Proceedings. International Conference on*. IEEE. 2002, pp. 514–519 (cit. on pp. 42, 63).

[72] Garrick STAPLES. „TORQUE resource manager“. In: *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*. ACM. 2006, p. 8 (cit. on p. 22).

[73] Achim STREIT. „The self-tuning dynP job-scheduler“. In: *Parallel and Distributed Processing Symposium., Proceedings International, IPDPS 2002, Abstracts and CD-ROM*. IEEE. 2001, 8–pp (cit. on p. 25).

[74] Richard S SUTTON, Andrew G BARTO, et al. *Introduction to reinforcement learning*. Vol. 135. MIT press Cambridge, 1998 (cit. on p. 82).

[75] Wei TANG, Zhiling LAN, Narayan DESAI, and Daniel BUETTNER. „Fault-aware, utility-based job scheduling on BlueGene/P systems“. In: *Cluster Computing and Workshops, 2009. CLUSTER'09. IEEE International Conference on*. IEEE. 2009, pp. 1–10 (cit. on pp. 19, 42, 53).

[76] TOP500.ORG. *TOP500 Supercomputer Sites*. `https://www.top500.org/`. Online; last access 30 november 2018. 2018 (cit. on p. 1).

[77] D. TRYSTRAM. „Scheduling parallel applications using malleable tasks on clusters“. In: *Proceedings 15th International Parallel and Distributed Processing Symposium. IPDPS 2001*. Apr. 2001, pp. 2128–2135 (cit. on p. 17).

[78] Dan TSAFRIR, Yoav ETSION, and Dror G FEITELSON. „Modeling user runtime estimates“. In: *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer. 2005, pp. 1–35 (cit. on pp. 47, 48).

[79] Deshi YE, Xin HAN, and Guochuan ZHANG. „Online multiple-strip packing“. In: *Theoretical Computer Science* 412.3 (2011). Combinatorial Optimization and Applications, pp. 233–239 (cit. on p. 17).

[80] Andy B YOO, Morris A JETTE, and Mark GRONDONA. „Slurm: Simple linux utility for resource management“. In: *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer. 2003, pp. 44–60 (cit. on pp. 2, 21).

[81] Jia YU and Rajkumar BUYYA. „A taxonomy of scientific workflow systems for grid computing“. In: *ACM Sigmod Record* 34.3 (2005), pp. 44–49 (cit. on p. 14).

[82] SN ZHUK. „Approximate algorithms to pack rectangles into several strips“. In: *Discrete Mathematics and Applications dma* 16.1 (2006), pp. 73–85 (cit. on p. 17).

[83] Dmitry ZOTKIN and Peter J KELEHER. „Job-length estimation and performance in backfilling schedulers“. In: *High Performance Distributed Computing, 1999. Proceedings. The Eighth International Symposium on*. IEEE. 1999, pp. 236–243 (cit. on p. 61).

Abstract

High-Performance Computing (HPC) platforms are growing in size and complexity. In an adversarial manner, the power demand of such platforms has rapidly grown as well, and current top supercomputers require power at the scale of an entire power plant. In an effort to make a more responsible usage of such power, researchers are devoting a great amount of effort to devise algorithms and techniques to improve different aspects of performance such as scheduling and resource management. But HPC platform maintainers are still reluctant to deploy state of the art scheduling methods and most of them revert to simple heuristics such as EASY Backfilling, which is based in a naive First-Come-First-Served (FCFS) ordering. Newer methods are often complex and obscure, and the simplicity and transparency of EASY Backfilling are too important to sacrifice.

At a first moment we explored Machine Learning (ML) techniques to learn on-line parallel job scheduling heuristics. Using simulations and a workload generation model, we could determine the characteristics of HPC applications (jobs) that lead to a reduction in the mean slowdown of jobs in an execution queue. Modeling these characteristics using a nonlinear function and applying this function to select the next job to execute in a queue improved the mean task slowdown in synthetic workloads. When applied to real workload traces from highly different machines, these functions still resulted in performance improvements, attesting the generalization capability of the obtained heuristics.

At a second moment, using simulations and workload traces from several real HPC platforms, we performed a thorough analysis of the cumulative results of four simple scheduling heuristics (including EASY Backfilling). We also evaluated effects such as the relationship between job size and slowdown, the distribution of slowdown values, and the number of backfilled jobs, for each HPC platform and scheduling policy. We show experimental evidence that one can only gain by replacing EASY Backfilling with the Smallest estimated Area First (SAF) policy with backfilling, as it offers improvements in performance by up to 80% in the slowdown metric while maintaining the simplicity and the transparency of EASY. SAF reduces the number of jobs with large slowdowns and the inclusion of a simple thresholding mechanism guarantees that no starvation occurs.

Overall we achieved the following remarks: (i) simple and efficient scheduling heuristics in the form of a nonlinear function of the jobs characteristics can be learned automatically, though whether the reasoning behind their scheduling decisions is clear or not can be up to argument. (ii) The area (processing time estimate multiplied by the number of processors) of the jobs seems to be a quite important property for good parallel job scheduling heuristics, since many of the heuristics (notably SAF) that achieved good performances have the job's area as input. (iii) The backfilling mechanism seems to always help in increasing performance, though it does not outperform a better sorting of the jobs waiting queue, such as the sorting performed by SAF.

Résumé

Les plate-formes de Calcul Haute Performance (de l'Anglais *High Performance Computing*, HPC) augmentent en taille et en complexité. De manière contradictoire, la demande en énergie de telles plates-formes a également rapidement augmenté. Les supercalculateurs actuels ont besoin d'une puissance équivalente à celle de toute une centrale d'énergie. Dans le but de faire un usage plus responsable de ce puissance de calcul, les chercheurs consacrent beaucoup d'efforts à la conception d'algorithmes et de techniques permettant d'améliorer différents aspects de performance, tels que l'ordonnancement et la gestion des ressources. Cependant, les responsables des plate-formes HPC hésitent encore à déployer des méthodes d'ordonnancement à la fine pointe de la technologie et la plupart d'entre eux recourent à des méthodes heuristiques simples, telles que l'EASY Backfilling, qui repose sur un tri naïf premier arrivé, premier servi (de l'Anglais *First-Come-First-Served*, FCFS). Les nouvelles méthodes sont souvent complexes et obscures, et la simplicité et la transparence de l'EASY Backfilling sont trop importantes pour être sacrifiées.

Dans un premier temps, nous explorons les techniques d'Apprentissage Automatique (de l'Anglais *Machine Learning*, ML) pour apprendre des méthodes heuristiques d'ordonnancement *online* de tâches parallèles. À l'aide de simulations et d'un modèle de génération de charge de travail, nous avons pu déterminer les caractéristiques des applications HPC (tâches) qui contribuent pour une réduction du ralentissement moyen des tâches dans une file d'attente d'exécution. La modélisation de ces caractéristiques par une fonction non linéaire et l'application de cette fonction pour sélectionner la prochaine tâche à exécuter dans une file d'attente ont amélioré le ralentissement moyen des tâches dans les charges de travail synthétiques. Appliquées à des traces de charges de travail réelles de plate-formes HPC très différents, ces fonctions ont néanmoins permis d'améliorer les performances, attestant de la capacité de généralisation des heuristiques obtenues.

Dans un deuxième temps, à l'aide de simulations et de traces de charge de travail de plusieurs plates-formes HPC réelles, nous avons effectué une analyse approfondie des résultats cumulés de quatre heuristiques simples d'ordonnancement (y compris l'EASY Backfilling). Nous avons également évalué des autres effets tels que la relation entre la taille des tâches et leur ralentissement, la distribution des valeurs de ralentissement et le nombre de tâches mises en calcul par *backfilling*, par chaque plate-forme HPC et politique d'ordonnancement. Nous démontrons de manière expérimentale que l'on ne peut que gagner en remplaçant l'EASY Backfilling par la stratégie SAF (de l'Anglais *Smallest estimated Area First*) aidée par *backfilling*, car elle offre une amélioration des performances allant jusqu'à 80% dans la métrique de ralentissement, tout en maintenant la simplicité et la transparence d'EASY Backfilling. La SAF réduit le nombre de tâches à hautes valeurs de ralentissement et, par l'inclusion d'un mécanisme de seuillage simple, nous garantissons l'absence d'inanition de tâches.

Dans l'ensemble, nous avons obtenu les remarques suivantes : (i) des heuristiques simples et efficaces sous la forme d'une fonction non linéaire des caractéristiques des tâches peuvent être apprises automatiquement, bien qu'il soit subjectif de conclure si le raisonnement qui sous-tend les décisions d'ordonnancement de ces heuristiques est clair ou non. (ii) La zone (l'estimation du temps d'exécution multipliée par le nombre de processeurs) des tâches semble être une propriété assez importante pour une bonne heuristique d'ordonnancement des tâches parallèles, car un bon nombre d'heuristiques (notamment la SAF) qui ont obtenu de bonnes performances ont la zone de la tâche comme entrée (iii) Le mécanisme de *backfilling* semble toujours contribuer à améliorer les performances, bien que cela ne remédie pas à un meilleur tri de la file d'attente de tâches, tel que celui effectué par SAF.