HAL
open science

Improving the Performance of Batch Schedulers Using Online Job Runtime Classification

Salah Zrigui, Raphael y de Camargo, Arnaud Legrand, Denis Trystram

To cite this version:

Salah Zrigui, Raphael y de Camargo, Arnaud Legrand, Denis Trystram. Improving the Performance of Batch Schedulers Using Online Job Runtime Classification. Journal of Parallel and Distributed Computing, 2022, 164, pp.83-95. 10.1016/j.jpdc.2022.01.003 . hal-03023222v2

HAL Id: hal-03023222

https://hal.science/hal-03023222v2

Submitted on 28 Feb 2022

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Improving the Performance of Batch Schedulers Using Online Job Runtime Classification

Salah Zrigui, Raphael Y. de Camargo, Arnaud Legrand, and Denis Trystram

Abstract—Job scheduling in high-performance computing platforms is a hard problem that involves uncertainties on both the job arrival process and their execution times. Users typically provide only loose upper bounds for job execution times, which are not so useful for scheduling heuristics based on processing times. Previous studies focused on applying regression techniques to obtain better execution time estimates, which worked reasonably well and improved scheduling metrics. However, these approaches require a long period of training data.

In this work, we propose a simpler approach by classifying jobs as small or large and prioritizing the execution of small jobs over large ones. Indeed, small jobs are the most impacted by queuing delays, but they typically represent a light load and incur a small burden on the other jobs. The classifier operates online and learns by using data collected over the previous weeks, facilitating its deployment and enabling a fast adaptation to changes in the workload characteristics.

We evaluate our approach using four scheduling policies on seven HPC platform workload traces. We show that: first, incorporating such classification reduces the average bounded slowdown of jobs in all scenarios, second, in most considered scenarios, the improvements are comparable to the ideal hypothetical situation where the scheduler would know in advance the exact running time of jobs.

1 Introduction

High-Performance Computing (HPC) platforms are fundamental instruments for many scientific and industrial fields, and supercomputers are becoming increasingly larger and more complex [1]. This evolution instigates the need for more adaptive and elaborate scheduling strategies. One approach is to develop sophisticated ad-hoc scheduling algorithms [2], [3], [4], [5]. However, such algorithms are often specific to a given scenario, non-generalizable, and too hard to be easily understood. Another more appealing alternative is to use more generic scheduling heuristics based on index policies, which are functions that compute ordering priorities based on job characteristics. Two notable examples are First Come First Served (FCFS), which orders the jobs based on their arrival times, and Shortest Processing time First (SPF) [6], which orders the jobs based on their runtimes. These heuristics are frequently combined with a backfilling mechanism, which allows some jobs to skip the queue if they do not delay the scheduling of the first job in the queue. An example is EASY-backfilling [7], which couples FCFS with backfilling. Backfilling mechanisms and some scheduling heuristics, such as SPF, require the actual execution time of jobs, which is unknown *a priori* in the majority of online scheduling scenarios. The scheduler has only access to user-provided upper-bounds, which are typically highly over-estimated [8]. Consequently, obtaining reasonable runtime estimates would be very valuable when designing HPC system schedulers.

- *S. Zrigui, A. Legrand and D. Trystram are with Univ. Grenoble Alpes, CNRS, Inria, Grenoble INP, LIG, 38000 Grenoble, France*
  Email: firstname.lastname@univ-grenoble-alpes.fr
- *R. de Camargo is with Federal University of ABC, CMCC, São Paulo, Brazil.*
  E-mail: raphael.camargo@ufabc.edu.br

Machine-learning techniques have emerged as a suitable tool to predict job execution times [8], [9], [10], [11]. However, it is difficult to estimate the execution times from historical data present in workload logs using regression-based techniques [12]. Such difficulty arises from the fact that crucial information, such as job dependencies, parameters, and even names, are often missing. Moreover, runtime information such as job placement and machine utilization are available only *a posteriori*. Consequently, although regression may allow better implementations of heuristics and tighter backfilling choices, obtaining reliable execution time estimates is rarely possible.

One insight one can have is that the key factor of heuristics that favor shorter jobs, such as SPF, is that executing small jobs first improves the metrics, such as the job flow time and the slowdown [13]. In this work, we follow this insight and we propose to apply a simple two-class classification instead of regression. We classify the jobs into two general classes, namely small and large, and prioritize the execution of small jobs. Since performing two-task classification is easier than full regression, we expect to obtain better classification performance with less training data. We perform a thorough evaluation using seven workload traces from actual HPC platforms and four scheduling policies, comparing the results of schedulers that use our job class classification with (i) standard schedulers, which rely only on user-provided information, and (ii) clairvoyant schedulers, which have perfect knowledge of actual job execution times. We show that:

- The a priori knowledge of whether the job is small or large is sufficient to generate scheduling improvements close to those obtainable using fully clairvoyant schedulers;
- Our online classification algorithm achieves a precision

ratio between 78% and 89% in all workload traces, which is sufficient to improve scheduling performance in all scenarios;
- Adding a safeguard mechanism that kills the jobs that are misclassified as small results in improvements similar to those obtainable using fully clairvoyant schedulers.

The remainder of this paper is organized as follows. We present a brief review of related works in Section 2. Section 3 contains the workload information and evaluation metrics used throughout this paper. Sections 3.3 and 4 describe the method used for the classification. Section 5 describes the experimental protocol, followed by the experimental evaluation (Section 6). Finally, in Section 7, we give some concluding remarks.

2 Related Work

Online Scheduling in HPC platforms is a hard problem that is plagued with many uncertainties. The evolving architecture of HPC platforms and the ever-changing nature of its users over time coupled with inaccurate runtime estimates makes attempting to determine a good scheduling scheme an elusive goal. To understand such uncertainties or at least try to circumvent them, many researchers have started evaluating the use of machine learning techniques. Throughout the literature, a wide range of learning-based solutions have been proposed. We distinguish two main approaches: (i) reducing the uncertainty in the scheduling data by adjusting job runtime estimates, and (ii) directly designing a scheduling scheme that improves specific objectives.

The first approach consists of using machine learning techniques to improve runtime estimates. Tsafrir *et al.* introduced EASY++, a variation of the classical EASY strategy, which replaces user-provided runtime estimates by the average runtime of the two previous jobs submitted by the same user [8]. Despite its simplicity, it allowed for improvement of around 25% over the classical EASY algorithm. Gaussier *et al.* used historical data from different traces and linear regression to predict runtimes with improved accuracy [9]. They also showed that predictions could be used more effectively if coupled with a more aggressive backfilling heuristic (namely SPF). Yet, they only focus on manipulating the backfilling policy (replacing FCFS with SPF) and do not explore the effects of changing the main index policy (FCFS). Later works [14], [15] show that the main ordering policy has a more significant impact on the general scheduling performance. A problem all the aforementioned prediction-based approaches frequently suffer from is the underestimation of running times. Guo *et al.* proposed a specific framework that can be used to detect runtime underestimates [10], allowing to adjust job running times accordingly. They compared their approach with classical prediction schemes such as SVMs and Random Forests and showed that it enhanced system utilization, but did not improve classical user-oriented metrics, such as those considered in this article.

An interesting phenomenon is that, increasing the inaccuracy (e.g., doubling the user-provided estimates) sometimes improves performance [16]. Such surprising behavior is related to Graham's scheduling anomalies and stems from the fact that index policies generally produce suboptimal scheduling. The policy used for scheduling has a major impact on the effectiveness of accurate predictions, with policies that favor shorter jobs benefiting more. More recent results by Gaussier *et al.* [9] show that, in some cases, predictions (which always have some inaccuracy) outperform their clairvoyant counterparts despite the latter's perfect knowledge of runtimes. During our previous studies, we also often encountered similar situations (especially when using workload resampling, which we avoid in the present work) but this remained an overall statistically insignificant effect.

In a recent study [12], the authors explored the effectiveness and limitations of using machine learning to improve the performance of computing clusters. They show that the workload is highly variable among periods, with large user churn and changes in machine utilization levels, and that a few users generate most workload. Consequently, model performance can vary strongly on a day-to-day basis. Moreover, more accurate runtimes do not systematically lead to better scheduling performance, and with the few datasets available today, it is difficult to assess the models performance. Finally, they argue that training can take many months (or years) before it reaches a stable level when using a few features, which would prevent practical deployments. We also observed strong day-to-day performance variability [15] and the potential inefficiency of static policies learned from long past periods of time. These observations motivate the need for reactive online learning policy that can quickly adapt to rapid load variations.

The second approach aims to design scheduling schemes that directly act on the ordering and allocation of jobs. Carastan-Santos and Camargo [11] used synthetic workloads and simulations to create index policy functions that improve the slowdown metric using non-linear regression. Interestingly, the generated functions resemble the Smallest Area First (SAF) policy. Legrand *et al.* [15] showed that using a linear combination of job characteristics allows building index policies that can significantly improve system performance. The authors also showed that the continuously changing nature of the data makes it very hard to learn online the optimal weights for this linear combination, preventing any static policy to be fully effective. Sant'Ana *et al.* [17] addressed the evolving nature of the workload by using machine learning techniques to select, in real-time, the best scheduling policy to apply for the next day on a given cluster, based on the current cluster and queue states. These attempts generated promising results but are rarely adopted by system administrators as they require deploying significant changes to existing scheduling policies. Also, some strategies rely on black-box scheduling algorithms.

The work presented in this paper falls under the first approach, focusing on managing the inaccuracy in the runtime estimates while using simple glass box scheduling algorithms. We propose classifying jobs into two classes, small and large, instead of performing regression-based execution time predictions. The objective is to allow faster training and adaptations to changes in the workload characteristics, while avoiding other issues, such as runtime underestimations.

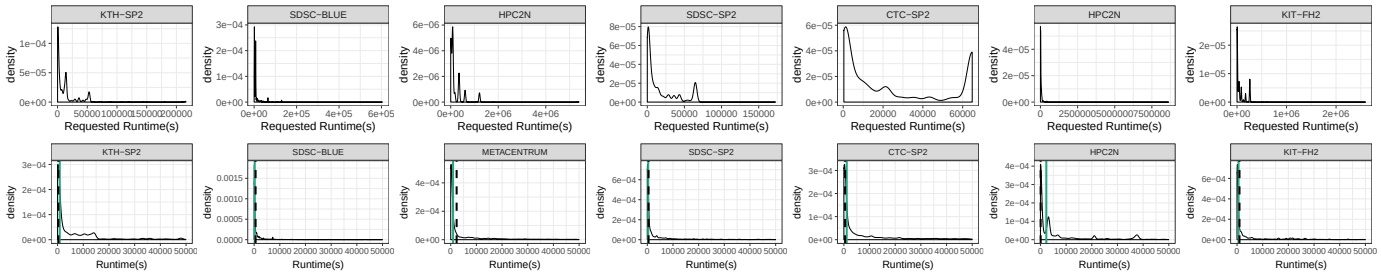

Figure 1. Distribution of requested (upper row) and actual (bottom row) execution times of jobs for the seven workload traces. (1) Note the scale/range difference on the X-axis, which indicates how the distribution of both variables are very different and shows that the requested runtime is a quite unreliable information. (2) The distribution of the actual runtime exhibits a sharp spike toward short jobs for all workloads. The green vertical line and the dashed black vertical line respectively represent the median value of the runtimes and the result of a clustering algorithm (Section 3.3) and allow to easily discriminate between "small" and "large" jobs.

3 Preliminary Observations

3.1 Workload Traces

We use a data-driven approach, which relies on the characterization and identification of workload patterns from execution logs (traces) of HPC platforms. To ensure that our approach can be generalized and is not specific to a particular cluster or machine, we used datasets from seven HPC platforms available from the Parallel Workloads Archive [18]. We show their main characteristics in Table 1.

TABLE 1
Workload traces characteristics

Name	Year	# CPUs	# Jobs	# Months
CTC-SP2	1997	338	77,222	11
HPC2N	2002	240	202,871	42
KTH-SP2	1996	100	28,476	11
SDSC-BLUE	2003	1,152	243,306	32
SDSC-SP2	1998	128	59,715	24
KIT-FH2	2016	24,048	114,355	19
MetaCentrum2-zegox	2013	576	79,546	24

In this work, we adopt the simple model of an HPC job as a rectangle, representing the runtime (width) and the number of requested resources (height). For each job j, we consider the following characteristics:

- The actual runtime p_j, which is known only after job completion;
- The requested runtime $\tilde{p}_j$, provided by the user at job submission. It is an upper bound of $p_j \leq \tilde{p}_j$ and is generally used as an estimate of p_j in scheduling heuristics;
- The number of requested processors q_j, which is static and provided by the user at job submission;
- The submission time r_j, also known only as release date.

Job runtime distributions change from one system to another, and building a unified runtime distribution model has proven to be a challenging task [19]. Nevertheless, the density of requested runtimes for all seven traces shows one or two peaks at small values, showing that most jobs have relatively small processing time requirements (Figure 1, upper row). Other peaks also appear in some traces, with some containing a peak near the maximum allowed processing time. However, when comparing to the *actual* runtimes (Figure 1, bottom row), we can easily see the well-known mismatch between the requested and actual runtimes. We also notice that the seven traces share an interesting similarity, with all actual runtime distributions having a sharp peak at the small values and a large tail towards longer execution times. These distributions indicate that we can always divide jobs into two classes: (i) small, encompassing the jobs at the peaks of the distributions, and (ii) large, comprising jobs at the tails of the distribution.

3.2 Evaluation Metrics

There exist several cost metrics, and each evaluates the performance of specific aspects of HPC platforms [20]. In this work, we focus on the *bounded slowdown* (*bsld*) metric, which represents the ratio between the time a job spent in the system and its running time. This ratio represents the slowdown perceived by the job when running on the system with all other jobs compared the situation where it would have had the whole system for itself only. It is defined as:

$$bsld_j = \max\left(\frac{wait_j + p_j}{\max(p_j, \tau)}, 1\right)$$

The value $wait_j$ is the time the job spent in the submission queue, p_j is the actual execution time, and τ is a constant that prevents very short job times from generating arbitrarily large slowdown values. We set τ to 60 seconds as it is commonly done. The reasoning behind the slowdown metric is that the response time and the waiting time of a job should be proportional to its runtime. In our study, we use the cumulative and the mean bounded slowdown, which are computed as the sum (resp. mean) of $bsld$ of all the jobs that have been executed so far. It is updated every time a job finishes its execution. It is well known that the mean bounded slowdown, which represents the responsiveness of the system, is generally optimized by giving a higher priority to short jobs. A strict prioritization comes at the risk of potential starvation of larger jobs but can be mitigated by fairness and reservation mechanisms.

3.3 Characterizing Small and Large Jobs

From now on, we consider a job as small if its runtime is smaller than the median of the runtimes (green line on Figure 1), which we call the (*divider*), and we consider the job as large otherwise. For the sake of comparison, we also applied two clustering algorithms, DBSN [21] and EM [22],

TABLE 2
Percentage of premature and non premature jobs: 22 to 49% of all jobs (Small premature jobs) requested their time allocation to be larger than the divider (5 to 20 minutes) but actually executed less than this

Trace	Divider (s)	Small non premature (%)	Small premature (%)
CTC-SP2	1,114	17.11	32.89
HPC2N	2,287	27.63	22.38
KTH-SP2	847	15.37	34.63
SDSC-BLUE	229	0.36	49.70
SDSC-SP2	359	12.02	38.01
KIT-FH2	601	2.42	29,96
MetaCentrum2-zegox	915	3.94	46.07

TABLE 3
Contribution of job size classes to platform resource usage: half of the jobs (Large) consume more than 98% of resources. Small jobs incur an insignificant workload and running them first (provided they can properly be identified) should thus be harmless to large jobs

Trace	Large (%)	Small non premature (%)	Small premature (%)
CTC-SP2	98.37	1.34	0.29
HPC2N	99.35	0.50	0.15
KTH-SP2	99.59	0.36	0.05
SDSC-BLUE	99.32	0.57	0.11
SDSC-SP2	98.33	1.45	0.22
KIT-FH2	99,79	0.008	0,20
MetaCentrum2-zegox	99.33	0.20	0.47

to divide the classes into two groups, which generated comparable divisions (dashed black line in Figure 1). Although divisions achieved by the median and clustering algorithms are not the same, they are relatively similar. As we generally aim for simplicity, we considered that the rolling median is sufficient to separate the initial peak from the rest of the distribution. Furthermore, having two classes with similar sizes simplifies the comparison in terms of fairness.

We further divide the small job class into two subclasses: (i) premature small jobs: short jobs that had requested runtimes larger than *divider* and which therefore terminate prematurely, and (ii) non-premature small jobs: those that also requested runtimes smaller than *divider* but managed to execute within this time bound. When analyzing the traces from the seven evaluated platforms, we notice that there is always a large fraction (22% to 50%) of premature jobs (Table 2). Premature small jobs have a wildly over-estimated processing time, causing them to wait longer for execution, which results in large slowdown values. This is problematic as it artificially inflates the overall slowdown. Moreover, we note that the total area[1] of these premature jobs represents a negligible fraction (less than 0.5%) of the total area (Table 3). If one could correctly detect these premature small jobs, we would obtain a significant reduction in the overall average slowdown in the platform. In the next section, we propose a method for performing this classification.

4 Job Size Classification

4.1 Classification Features

In this section, we detail the features we used for the classification and the reasoning behind our choices. A job is characterized by a set of *features*, which are pieces of information that can be used to predict the *class* of the jobs (Small or Large in our context). When a job is submitted, the scheduler has access to the following information: the id of its user,

1. The area a_j of a job j is defined as its runtime multiplied by the number of resources it requested: $a_j = p_j * q_j$.

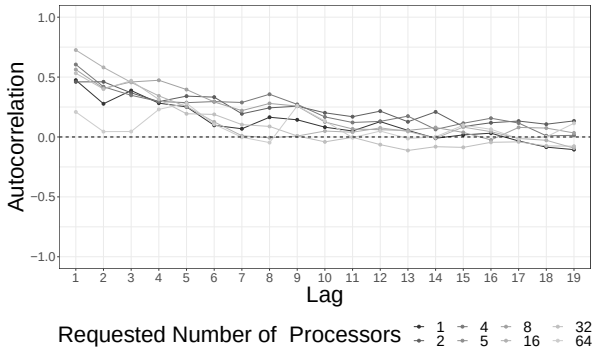

Figure 2. Each category c allows to extract a series (ordered by submission dates) of actual runtimes $p^{(c)}$ for which we can estimate the autocorrelation coefficient for each lag value l as follows: $\rho_{p^{(c)}}(l) = \frac{\frac{1}{n-l}\sum_{i=1}^{n-l}\left(p_i^{(c)} - \mu_{p^{(c)}}\right)\cdot\left(p_{i+l}^{(c)} - \mu_{p^{(c)}}\right)}{\sigma^2_{p^{(c)}}}$, where $\mu_{p^{(c)}}$ and $\sigma_{p^{(c)}}$ are respectively the sample average and sample standard deviation of $p^{(c)}$. This autocorrelation coefficient lies in $[-1, 1]$ and indicates how strongly $p_i^{(c)}$ is correlated with $p_{i+l}^{(c)}$. The graph illustrates how the distribution of the autocorrelation coefficient evolves with the lag between the jobs that belong to the same category (u, q) of a specific user.

the dimensions of the jobs (requested number of processors, requested runtime), and the exact date of submission. We start by making two observations about the scheduling data: (i) it has been empirically observed that the runtime of a job is highly correlated with the user's submission history [12]; (ii) although there are clear regularities, the user identity is not sufficient to characterize job duration because the users often submit more than one *category* of job. For example, user_2 of the SDSC-SP2 submitted 796 jobs with 8 different requested node numbers and 11 different requested runtime values[2]. For a given user, the requested runtime of jobs, their size and the day when they are submitted are however a good indicator of the similarity of their actual runtime. We therefore introduce a category for each pair of factors (u, q), $(u, \tilde{p})$, and (u, d) and consider that two jobs from the same user u belong to *category* (u, q) (resp. $(u, \tilde{p})$, or (u, d)) if they have the same number of requested resources q (resp. same requested runtime $\tilde{p}$, or same submission day d).

To illustrate how useful such categories could be, let us come back to the user_2 of SDSC-SP2. Figure 2 illustrates how the autocorrelation coefficient decreases with the lag l: jobs that are very close in time have a relatively strong correlation and the first few lags have significantly higher correlation values than the rest. The notion of category therefore structures the job flow and can be used to perform online prediction of the jobs actual duration.

For each job, we extract all previous jobs that belong to the same categories and we derive the following features (see Table 4.1):

Job features : The requested execution time and requested number of resources of the job.

Temporal features : The hour of the day, the day of the week, the month and the quarter in which a job was

2. Although $\tilde{p} \in \mathbb{R}^+$ may be arbitrary, in most HPC environments users tend to restrict themselves to a finite and small set of round values (e.g., 1 hour). This value can thus be treated as a factor.

TABLE 4
Features used for job classification

Type	Feature	Description
Job features	$\tilde{p}_i$	user supplied runtime estimate
	q_i	user supplied number of resources
Temporal features	h	hour of the day
	D_{week}	day of the week
	d_{month}	day of the month
	m	Month
	w	Week
	Q	Quarter
Lag features	$C\tilde{p}_{i-1}$, $C\tilde{p}_{i-2}$, $C\tilde{p}_{i-3}$	Class of the previous, second to previous, third to previous jobs that belong to the same category $(u, \tilde{p})$
	Cq_{i-1},Cq_{i-2}, Cq_{i-3}	Class of the previous, second to previous, third to previous jobs that belong to the same category (u, q)
	Cd_{i-1},Cd_{i-2}, Cd_{i-3}	Class of the previous, second to previous, third to previous jobs that belong to the same category (u, d)
Aggregation features	$mean_{i\tilde{p}}$	percentage of jobs that belong to the same category $(u, \tilde{p})$ and are classified as small
	$mean_{iq}$	percentage of jobs that belong to the same category (u, q) and are classified as small
	$mean_{id}$	percentage of jobs that belong to the same category (u, d) and are classified as small

submitted;

Lag features : contains the class (Small/Large) of the previous three jobs of the same category;

Aggregation feature : contains the percentage of jobs that belong to the same category and are classified as small. The goal of these features is to include the rest of the category's history. Although older jobs are less indicative of the class of the current job, they still contain information that is valuable to the learning process.

4.2 Classifier Training and Update

In an online scheduling context, the full information about the jobs is only known after their execution. Thus, the classical learning scheme, which consists in dividing the full dataset into a training and a testing set is not possible in this context. The learning process should adapt to the increasing amount of available data. We adopt a weekly training process illustrated in Figure 3:

- Training is performed at the end of every week.
- All the data gathered during the week is cleaned and processed to create the features presented in Table 4.1.
- A new classifier is then trained and will be used during the next week.

We chose the period of one week because it seemed adequate. Indeed, retraining every day would be wasteful because in most cases a single day is not sufficient to generate enough new data to significantly change the output of the training. And retraining when the size of the new data reaches a certain threshold (e.g., training every 5000 new jobs) would cause the classifier to be updated at "unpredictable time", possibly in the middle of a workload spike, which is quite undesirable.

The data of the current week jobs are thus not added to the training data since we perform a weekly training. Note that, to simplify the implementation of our simulations and the tuning of the learning algorithms, we have decoupled the batch scheduling simulation from the learning and prediction mechanism. As a consequence, when performing predictions, the lag and the aggregation feature are also solely extracted from the jobs of previous weeks, which slightly decreases the reactivity of our predictions as they are built on information that is not the most up-to-date.

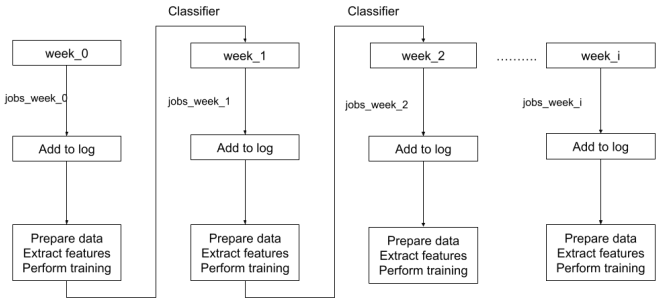

Figure 3. Learning process: At the end of each week the new jobs are added to the dataset and a new training process is performed

We use the Random Forest algorithm [23] to perform the classifications as it allows to easily combine numerical and categorical features. Random forests create multiple decision trees on randomly selected data samples, getting a prediction from each tree, and select the best solution by majority voting, which makes them particularly resilient to "outliers".

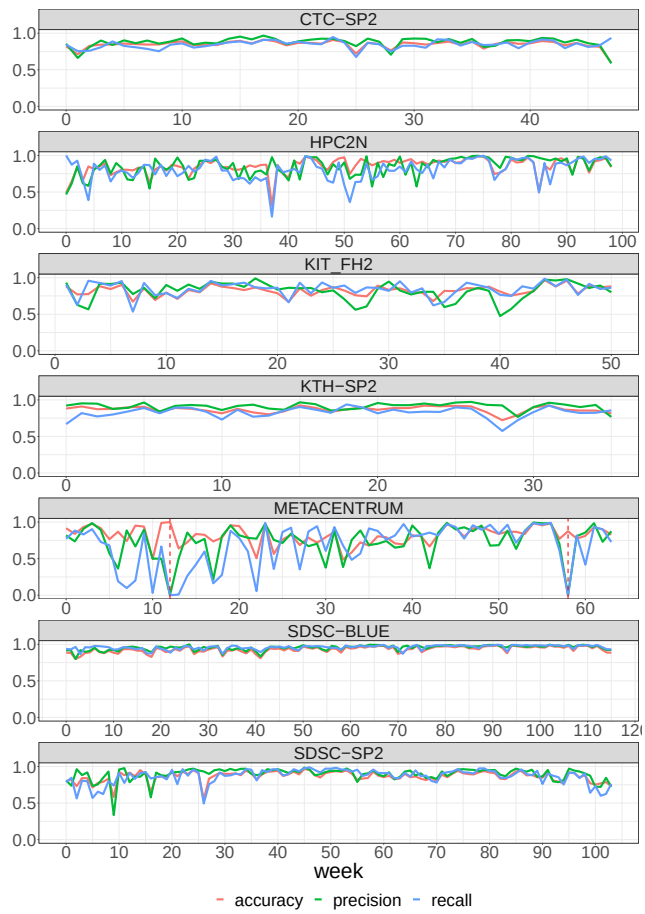

Figure 4. Evolution of the quality of the learning for individual weeks

4.3 Online Learning Quality

In this section we investigate the quality of the proposed online classification scheme and we explore some of the strengths and weaknesses while applying learning on scheduling data.

4.3.1 Accuracy, Precision and Recall

We measure the quality of our classifications using the three following indicators[3]:

- *Accuracy* is the ratio of correctly predicted observation over the total number of observations:

$$accuracy = \frac{TL + TS}{TL + TS + FL + FS} \quad (1)$$

- *Precision* is the ratio of correctly predicted small jobs to the total number of jobs that are predicted to be small:

$$precision = \frac{TS}{TS + FS} \quad (2)$$

- *Recall* is the ratio of correctly predicted small jobs to all observations in the small class:

$$recall = \frac{TS}{TS + FL} \quad (3)$$

As explained in section 4.2 the learning process is repeated at the end of every week and every week may thus have a different classification performance. Some week may suddenly behave very differently from the previous ones and thus, it is interesting to study the process in more detail. Figure 4 depicts the weekly quality of the learning through time for each platform, and from which several observations can be made.

Although the quality of the learning process varies between traces, which is expected because every trace has its own specific characteristics (number of jobs, number of users, frequency of job arrival, etc.) and may be more or less variable, for CTC-SP2, KTH-SP2, SDSC-SP2, and SDSC-BLUE all the weekly evaluation metrics maintain high values from the beginning to the end (with a few exceptions). For the other two traces; MetaCentrum and HPC2N, the results are not as stable as the others over time. For several weeks of MetaCentrum, the learning even seems to be very poor as the precision and recall values are extremely low and even drop to 0 for some cases. A closer look at these weeks allows to understand why such low values occur.

Week 58 (identified by the rightmost red dashed line in the MetaCentrum trace of Figure 4) comprises 48 jobs, only 1 of which is a small job. The classifier in this instance made a single error and misclassified this single small job as large. This leads to a significant reduction in the accuracy value; 85% (due to the small number of jobs in that week), a 0% recall, and an undefined value for the precision ($TS = 0$ and $FS = 0$)

Week 12 (identified by the leftmost red dashed line in the MetaCentrum trace of Figure 4) comprises only 78 jobs but 0 small jobs and all the jobs are properly classified, which results in an accuracy of 100% but the values of the precision and recall (Equations (2) and (3)) cannot be computed because $TS = 0$ and $FS = 0$ and $FL = 0$.

The misclassification of a single job may thus significantly impact the learning metrics for a week comprising few jobs but it has a relatively low impact on a trace of more than 79,000 jobs (Table 1). These absolute fine grain weekly learning performance indicators should be interpreted with care, especially because some weeks with a relatively low workload or missing classes (e.g., week 30 of MetaCentrum) often make the learning look artificially inefficient.

3. TL –True Large– (resp. TS) represents the number of jobs correctly predicted as Large (resp. Small), while FL –False Large– (resp. FS) represents the number of jobs incorrectly classified.

TABLE 5
General classification performance: For each trace, we count the values of TS,FS,TL and FL for all the weeks then, we compute the general value of the accuracy, precision and recall

Trace	Accuracy(%) $TL+TS/Total$	Precision(%) $TS/(TS+FS)$	Recall(%) $TS/(TS+FL)$
CTC-SP2	85	82	86
HPC2N	90	87	89
KTH-SP2	86	79	90
SDSC-BLUE	80	78	83
SDSC-SP2	87	89	91
KIT-FH2	80	90	77
MetaCentrum2-zegox	85	83	87

TABLE 6
Classification error per trace: 7–11% of Large jobs are misclassified while 4–8% of Small jobs are misclassified

Trace	False Large(%) $FL/(FL+TL)$	False Small(%) $FS/(FS+TS)$
CTC-SP2	8.16	6.29
HPC2N	7.59	5.05
KTH-SP2	9.27	4.03
SDSC_BLUE	11.29	8.52
SDSC-SP2	7.20	5.85
KIT-FH2	14.2	5.01
MetaCentrum2-zegox	8.43	6.36

Fortunately, the overall (i.e., when the ratios of Equations (1) to (3) are not broken down per week) performance of the classifier is particularly good. Table 5 shows the performance of all the jobs regardless of the week. The overall recall, precision and accuracy are always above 78% and even generally around 90%.

4.3.2 Classification Errors

During the training phase, the goal is to reduce prediction errors as much as possible. However, incorrect predictions and errors are an unavoidable part of any learning process. The two types of prediction errors are related to the number of false large (FL) and false small (FS) jobs. Table 6 shows the percentage of classification errors of each type. Although the values vary from one trace to another, the percentage of FS tends to be smaller than that of FL. We note that the percentages of FS jobs we obtained are comparable to the values presented in [10] where the authors use a method to specifically manage the problem of underestimation in runtimes predictions and reported an FS rate of $\approx$ 5–8%. We do not aim at designing a perfect classifier nor at fine-tuning the learning algorithm parameters so we argue that such classification error is representative of what can be achieved with a reasonable effort. Although the predicted class is a precious qualitative information, a scheduler should thus be aware of potential prediction errors and manage them accordingly.

4.4 Feature importance analysis

In this section, we provide insight on the importance and the stability of each of the features during the learning process. We use the Gini impurity measure [24], which estimates the probability of miss-classifying an observation, to evaluate

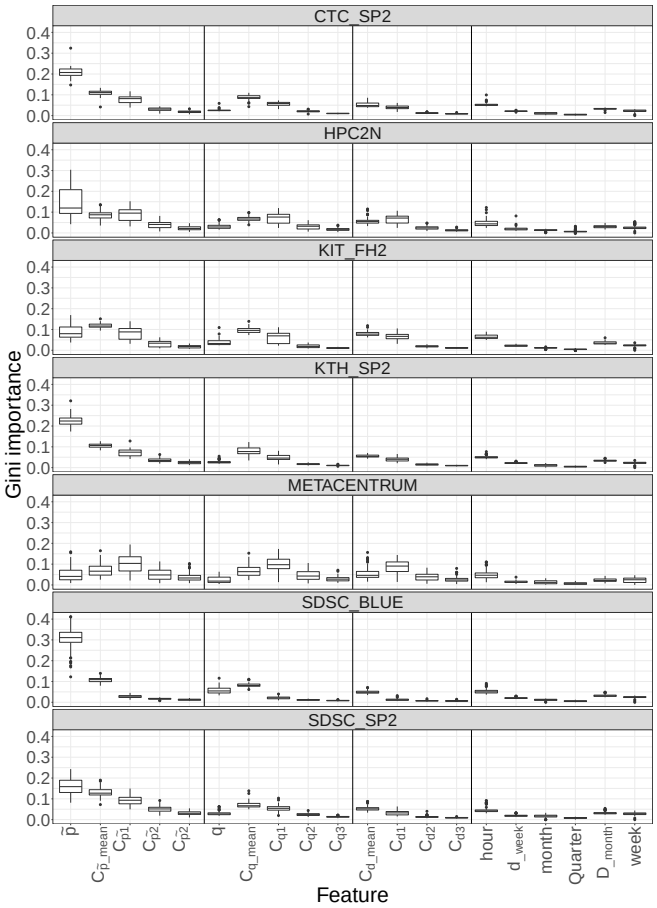

Figure 5. Importance of individual features during the weekly learning process. The larger the weights, the more important the feature in the classification. Weights are normalized such that their sum equals 1.

the importance of the constructed features. Note that it is also the measure used during the training phase of our classification. For an in-depth review of feature importance analysis, we refer the reader to [25]. This measure provides us for each week with a weight representing the importance of each feature in the classification.

Figure 5 represents the distribution (summarized with a box-plot) of the weekly weights of each feature for each trace. These box-plots reveal several useful information on the learning process.

- First, we observe similarities between the results of various traces. (i) The requested execution time appears to be the most important feature for the majority of the traces (expect METACENTRUM). This is expected since a sizable portion of the jobs belong to the class *Small* because the users themselves requested a processing time that is smaller than the divider (see table 2). (ii) For the lag features, the first lag always holds most of the weight followed by the second and the third lag respectively. (iii) Aggregation features are about as important as the first lag features. (iv) Temporal features generally hold the lowest weights for all the traces.
- This supports the idea that performing a single unified learning process for all traces like the ones presented in [26] and [11] is reasonable and can yield good results. However, there are also clear differences between the traces. For example, the requested time, the feature with the highest weight, holds different importance scores from one trace to another and it is outweighed by the lag features in the case of the METACENTRUM trace. Also, for some traces (CTC-SP2 and KTH-SP2), we observe a very small difference between the weeks. For others (METACENTRUM and HPC2N), there is a noticeable difference between the weeks, which can be explained by the fact that this workload seems harder to predict than the others and supports the importance of constantly updating the learning process.

5 Proposal

As indicated by the previous trace analysis, small jobs represent a negligible fraction of the total load of the platform (Table 3) but are quite numerous (Table 2) and often wildly over-estimate the runtime they request. Scheduling them using a policy based on this estimation leads to particularly poor slowdown and to an overall poor performance of the system. Fortunately, the learning algorithm presented in Section 4 allows to efficiently distinguish between small and large jobs. To improve the mean bounded slowdown, we propose that any job classified as small is executed with a higher priority than those classified as large. Specific care must be taken though to avoid starvation and to deal with classification errors. This section describes how this was done and how we evaluated our proposal.

We aimed our work to be as transparent and reproducible as possible [27]. We provide a snapshot of the workflow used throughout this work[4], which includes a nix [28] file that describes all the software dependencies and an R notebook that allows reproducing all the figures.

We consider HPC platforms as a collection of homogeneous resources, with jobs stored in a centralized waiting queue, following the submission described in the workload logs. We implemented all simulations using Batsim [29], a simulator based on SimGrid [30] that allows us to observe the behavior of scheduling algorithms under different conditions. We evaluate our method using the seven traces presented in Table 1 and four scheduling policies presented in the following section.

5.1 Scheduling Policies

We considered four scheduling policies:

- **FCFS:** First Come First Served [7] orders the jobs by their arrival time r_j. FCFS is the most commonly used scheduling policy.
- **WFP:** is a scheduling policy adopted by the Argonne National Labs [31] and is given by: $WFP_j = (\frac{wait_j}{\tilde{p_j}})^3 * q_j$. This policy attempts to strike a balance between the number requested resources, the requested runtime and the waiting time of jobs. It puts emphasis on the number of requested resources while preventing small jobs from waiting too long in the queue.
- **SPF:** Shortest (requested) Processing time First [6] orders the jobs by the requested processing time ($\tilde{p_j}$) given by the user.

4. https://gitlab.inria.fr/szrigui/job_classification/

- **SAF:** Smallest Area First [13] orders the jobs by their requested area $\tilde{a_j} = \tilde{p_j} * q_j$.

We chose FCFS and WFP because several existing HPC systems use them. SAF and SPF are less common, mostly because they are perceived as too risky since they could potentially induce job starvation. *Starvation* occurs when a job waits for an indefinite or a very long time in the queue. However, some studies [13], [15] show that SAF and SPF provide better results on performance metrics in almost all cases. Furthermore, one can prevent starvation by putting a *threshold* on the waiting time [14]. When the waiting time of a job surpasses the threshold, the scheduler transfers the job to the head of the queue. In [14] the authors perform a detailed analysis of the thresholding mechanism and of its impact.

We implemented the four aforementioned policies in conjunction with the EASY backfilling heuristic and the thresholding/starvation prevention mechanism. The scheduler orders and executes the jobs following the order set by the chosen policy. When it reaches a job that cannot start immediately, it makes a reservation for that job. The scheduler then allows the next tasks to skip the queue if they do not delay the initial reservation.

5.2 Learning and Scheduling Algorithms

When a user submits a job for execution, the classifier uses the job features to assign it to the small or large classes, represented by queues Q_{small} and Q, respectively. In the first week of the trace, since the classifier does not have prior data to learn the classification task, it classifies all jobs as large and does not behave differently from a classical policy. After that, we update the classifier at the beginning of every week, with data from all previous weeks as explained in Section 4.

The resource manager calls the scheduler whenever a job finishes its execution, and computational resources become available. The scheduler then sorts the two queues, Q and Q_{small}, independently, according to a chosen policy (FCFS, SAF, SPF, or SQF), and merge them in a single queue Q_{total}, with the jobs belonging to the small class first. Finally, resource allocation is done using the EASY heuristic, as shown in Algorithm 1.

Algorithm 1: Perform scheduling

Input : Queue of large jobs Q
Queue of small jobs Q_{small}
Scheduling policy $Policy$ (FCFS|WFP|...)

1 Order Q according to $Policy$
2 Order Q_{small} according to $Policy$
3 $Q_{total} =$ **concat**(Q_{small}, Q) # *small jobs are always put in the head of the final queue*
4 **EASY**(Q_{total}) # *Schedule the jobs in the final queue using the EASY heuristic*

The only additional relevant overhead compared to the basic EASY scheduling heuristic is the cost of updating the classifier. The update includes finding the median execution time over the workload log of the previous week and training the classifier using the pairs $(features, jobclass)$.

Algorithm 2: Kill False Small jobs

```
1  Q = {} # queue of large jobs
2  Q_small = {} # queue of small jobs
3  foreach job_j in Running do
4      # go through all jobs currently running
5      if job_j.class == "small" & job_j.runtime > divider
       then
6          kill(job_j)
7          Q_small.remove(job_j)
8          Q.add(job_j)
9      end
10 end
```

The full execution of this procedure takes only a few seconds and occurs only at the end of every week. Moreover, it runs independently from the scheduler, without blocking it.

5.3 Dealing with Classification Errors

As explained in Section 4.3.2, no classifier is perfect and some jobs will inevitably be wrongly classified. False Small jobs are large jobs that were wrongly classified as small. This is similar to runtime underestimation in the case of exact runtimes prediction. Although the resource manager may allow these jobs to execute until completion, it can significantly impact performance in some cases, e.g, when there are many True Small jobs following the misclassified large job. This type of misclassification is arguably more dangerous than False Large: if a small job is classified as large it will be delayed but it will not impact the waiting time of other jobs.

One way to correct this problem is to kill false small jobs. When the execution time of a job classified as small exceeds the divider value, it is killed and put back to the waiting queue as a large job. To ensure that the killing and restart process happen without problems, we consider that jobs are **idempotent**. Formally an idempotent operation is defined as an operation that can be applied multiple times without changing the result from the initial application. In this context, we consider a job to be idempotent if it can be killed and restarted without changing the final execution outcome.

The scheduler periodically goes through the list of running jobs (Algorithm 2). If it detects a job classified as small and has been executing for a period longer than the divider value, it kills the job and classifies it as large.

6 Experimental Results

In Section 4, we presented the job size classifier and showed its accuracy from a pure learning perspective. However, achieving a high-quality classification is not our final goal. In the scheduling context, the effectiveness of an approach is measured by how much it improves the end-to-end performance metrics, such as the average bounded slowdown (Section 3.2).

6.1 Overall Impact on Scheduling Performance

We evaluate the impact of the cumulative bounded slowdown when applying the EASY-backfilling with the four

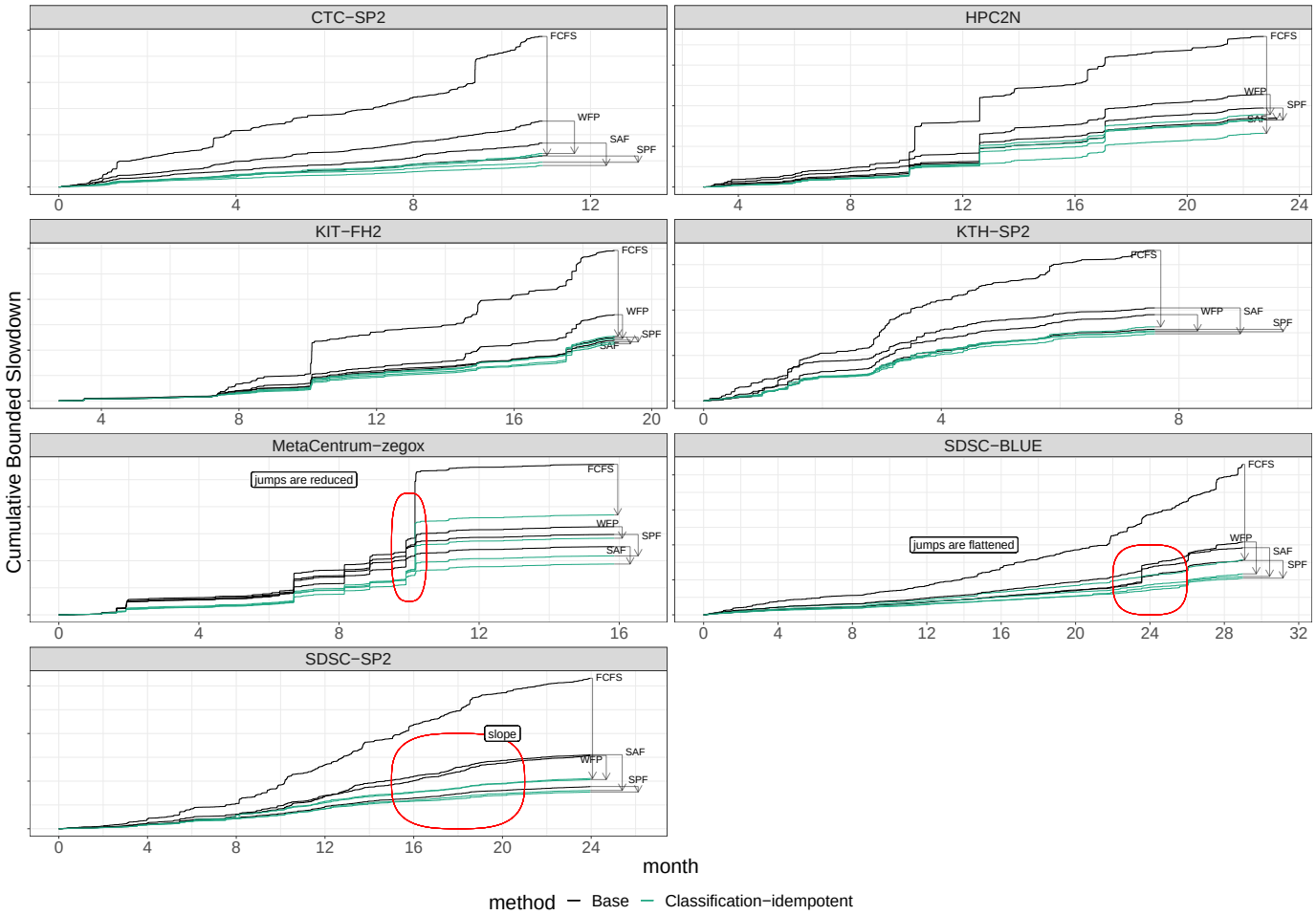

Figure 6. Evolution of the Cumulative Bounded Slowdown for the seven platforms, using the base policies (black) and the same policies augmented with job size classification and idempotence (cyan). The cumulative bounded slowdown is always such that SAF≈SPF<WFP<FCFS, which is expected as prioritizing small jobs is known to optimize the average slowdown whereas FCFS rather bounds the largest waiting time. Since these heuristics solely rely on the requested runtime, they cannot be very efficient. Activating our classification-based prioritization systematically and significantly improves the performance of all heuristics at any point in time and not simply at the end of the evaluation period. In steady state (see SDSC-SP2), it is clear that the cumulative bounded slowdown increases more slowly when our classification-based mechanism is activated. It may happen that bursts of jobs are submitted and incur sudden and large jumps in the cumulative bounded slowdown. These jumps are always significantly reduced (see Megacentrum-zegox) with our mechanism and even sometimes completely avoided (see SDSC-BLUE).

scheduling policies (FCFS, WFP, SAF, and SPF). Figure 6 shows the results for the scenarios with the job size classification and job-killing mechanism (in cyan) and without them (in black).

Comparing the curves for the four basic scheduling policies, we note that SPF and SAF generate the lowest cumulative slowdown in all platforms. WFP has cumulative values close to SPF and SAF, while FCFS has the worst values by a large margin in all cases. These results are consistent with previous comparisons of scheduling policies [14], [15].

Applying the job size classification reduced the cumulative slowdown values in all scenarios, with the improvement depending on the trace and scheduling policy. For FCFS, we observed substantial improvements for all the seven traces, ranging between 33% to 79%. For the other policies, we observed smaller, albeit consistent, improvements in performance, ranging from 3% to 32% for SPF and 10% to 51% for SAF. We explore these results further in Section 6.5.

The cumulative slowdown increases most of the time smoothly, with some sharp rises. The slower increments occur during lightly or moderately loaded periods, in which we see steady increments in the gap between the scenarios with and without job size classification. The jumps are the result of high load periods and seem unavoidable, as they occur with all policies. However, regardless of the policy and the trace used, our method always results in smaller cumulative slowdowns.

Since FCFS performed poorly compared to other policies, we decided to exclude it from the subsequent analysis. However, we note that the observations in the next sections also apply to FCFS.

6.2 Impacts on Individual Months

The evolution of the cumulative bounded slowdown over long periods, although informative, can mask important details about the behavior of a scheduler at a smaller time scale, such as individual weeks or months. Ideally, a good scheduler should provide improvements that are somewhat equitably distributed throughout the evaluation period.

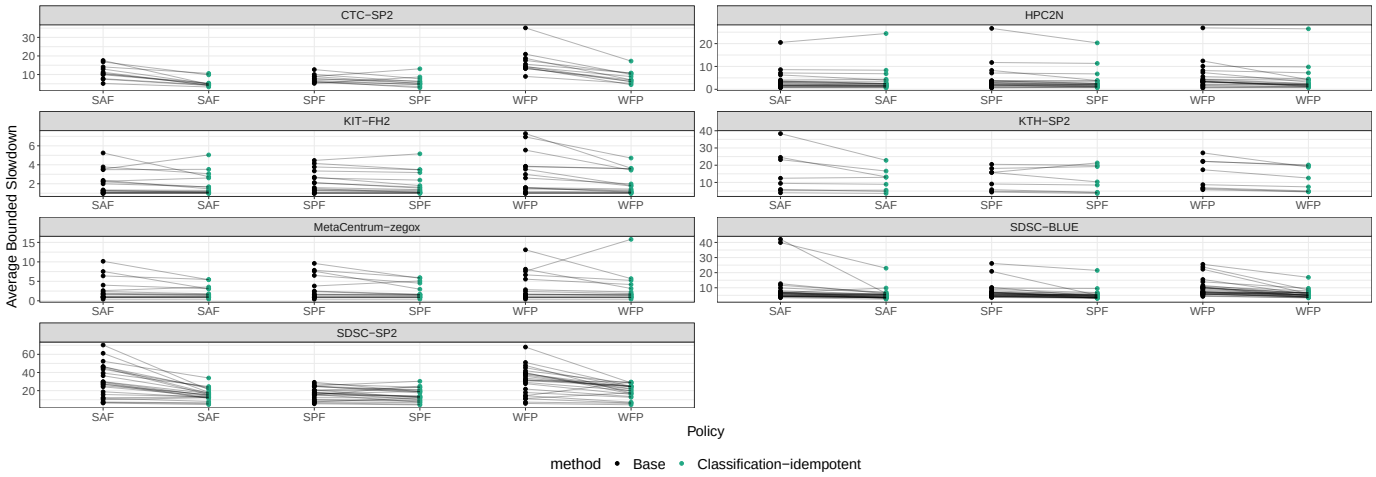

Figure 7. Monthly average bounded slowdown. Each line links the values from the same month when using the base and classification-idempotent schedulers.

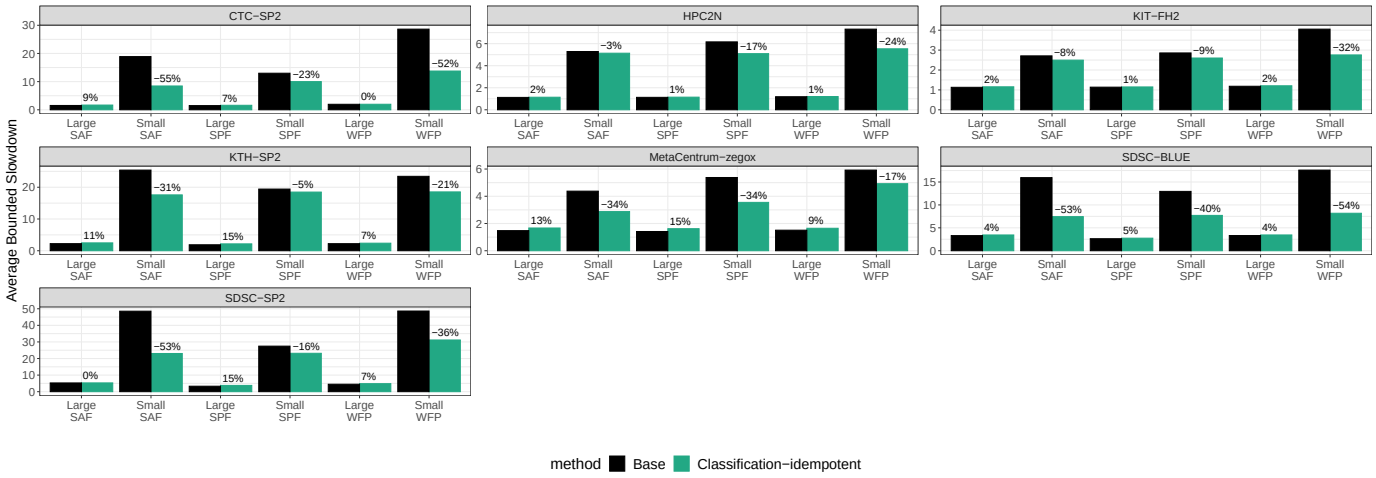

Figure 8. Average bounded slowdown for small and large jobs, using the four base policies and the corresponding classification-idempotent schedulers. Breaking down the average bounded slowdown between small and large jobs allows to evaluate how both classes benefit from the classification and whether one is unfairly treated compared to the other. The benefit for the Small job class is huge and can go up to 55% while the loss for the Large job class never exceeds 15% (the higher losses always occur in trace/policies with extremely small base slowdown). The difference for Large jobs is therefore negligible and would be barely noticeable by users. Last, note that, although there are visible differences between the base policies (SAF, SPF, WFP, in black), they tend to vanish whenever using our classification (in green).

We investigate the effects of our approach on individual months in Figure 7. Each pair of connected points represents the average bounded slowdown of a single month from the full workload execution, for the *base* and *classification-idempotent* cases. We note a reduction in the slowdown in most cases, with a decrease proportional to the base value. There are a few months where our approach seems to degrade performance substantially, such as in MetaCentrum/WFP. These are artifacts that emerge from splitting the results into one month periods, where workloads "spills" from one month to the other during periods of very high load. Overall, the results show that improvements are fairly distributed between months, even for the clusters that have large jumps in the cumulative slowdown, such as MetaCentrum and HPC2N.

6.3 Impact of Small Job Prioritization over Large Jobs

Our algorithm reduces the overall bounded slowdown by prioritizing small jobs. This mechanism naturally raises one crucial question: What is the impact of favoring small jobs over the jobs classified as large?

We compute the average bounded slowdown of the jobs for each of the two classes (Figure 8). As expected, the small jobs have the most substantial reductions in the average slowdowns. The extent of the reduction depends on the platform and policy and is mostly proportional to the improvements in the cumulative bounded slowdown, shown in Figure 6. More importantly, there is only a small increase in the average slowdown of large jobs.

The use of the job size classifier results in substantial improvements for small jobs, with minimal impact on large jobs.

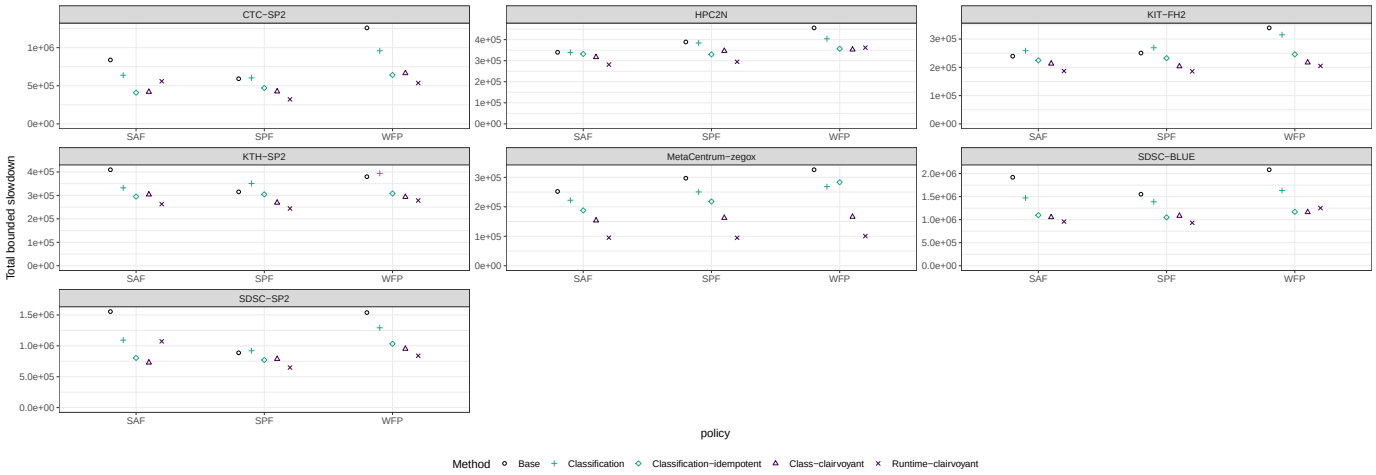

Figure 9. Total accumulated bounded slowdown for the base schedulers (base), schedulers with perfect classification (class-clairvoyant), schedulers with classification and job-killing mechanism (classification-idempotent), and schedulers with perfect execution time information (runtimes-clairvoyant). Regardless the heuristic, it is interesting to note that, in general, Base $\gtrapprox$ Classification $>$ Classification-Idempotent $\gtrapprox$ Class-clairvoyant $>$ Runtime-clairvoyant, which is consistent with the fact that more accurate information allow to produce better schedules

6.4 Impact of the Safeguard Mechanism

Assigning a large job to the small class can cause an overall increase in the average bounded slowdown of other jobs since it occupies resources for an extended period. We prevent this problem by killing the job when it reaches the job size divider value. But we cannot apply it for non-idempotent jobs. A subsequent question that arises is: can we still improve the performance if we allow miss-classified jobs to run until completion?

We compared the cumulative bounded slowdown values at the end of the full workload trace simulations, for the seven platforms, for three scenarios: (i) *base*, (ii) *classification-idempotent*, where we kill false small jobs, and (iii) *classification*, where we use classification without job-killing.

Preventing job-killing reduces the effectiveness of the classification in almost all scenarios (Figure 9). We note, however, that *classification* without job-killing still managed to improve the total slowdown for most cases, but to a worse extent than *classification-idempotent*. The exceptions are the combinations where the *classification-idempotent* only managed to improve results by a small margin. In these cases, the *classification* without job-killing did not improve or caused a negligible degradation in performance. Removing the safeguard mechanism reduces the effectiveness of our method without rendering it completely useless. Note that only False Small jobs are restarted, which corresponds to 2-4% of all jobs at most (see Table 6).

6.5 Comparison with Clairvoyant Schedulers

Finally, we evaluate the hypothetical optimum obtainable by a clairvoyant scheduler that knows the actual execution times of each job in advance. We compare three strategies that build the base policies (SPF, SAF, and WFP): (i) *runtimes-clairvoyant*, where the scheduling heuristic is provided with the actual p_j, instead of the requested processing times $\tilde{p}_j$, (ii) *class-clairvoyant*, where the scheduler is indicated which class the jobs belong to (i.e., as if a perfect job class classification was achieved), and (iii) *classification-idempotent*, the method we propose and which only uses estimated execution times. Although the clairvoyant versions cannot occur in practice, they provide us with a useful benchmark on the achievable improvements.

Using the *classification-idempotent* results in improvements comparable to the *class-clairvoyant* (Figure 9), except MetaCentrum. This result indicates that the job-killing mechanism is effective in counteracting the misclassifications and that the overhead of job-killing has a small impact on performance. Moreover, it shows that our strategy of combining classification with job-killing is already very efficient and has little room for further improvements.

The two clairvoyant strategies, *class-clairvoyant* and *runtimes-clairvoyant*, also have comparable performance, with slightly better results when using *runtimes-clairvoyant*. This result shows that a simple classification in two categories is, in most cases, sufficient to obtain important improvements for the bounded slowdown metric. It indicates that trying to predict job execution time accurately with elaborate regression techniques will not bring large improvements over the use of a simpler binary job size classification.

The most notable exception to the conclusions above is the MetaCentrum trace. We observe consistent improvements when moving from *base* to *classification-idempotent*, *class-clairvoyant*, and *runtimes-clairvoyant*. For this particular trace, there were several jumps in the cumulative bounded slowdown (Figure 6), caused by abnormally high loads. In these situations, a perfect knowledge of execution times appears to have a larger impact on scheduling performance.

Finally, we look at the cases were *class-clairvoyant* provided minor improvement: SDSC-SP2/SPF and KTH-SP2/SPF. In Figure 9, we can see that even with full knowledge, there were no significant improvements. *Class-clairvoyant* only improved over *base* SPF by 10% and 13% for SDSC-SP2 and KTH-SP2 respectively indicating that, for these two traces, SPF was already a very good policy.

TABLE 7
Improvement (in %) over EASY-FCFS using regression ([8] and [9]) and classification (SPF-CI and FCFS-CI). Values between brackets correspond to the evaluation performed by the original authors whose methodology may slightly differ from ours. Our classification based approach systematically and significantly improves upon the previous strategies, regardless of the the base scheduling heuristic (FCFS or SPF)

			Classification-Idempotent	
	EASY++ [8]	Gaussier et. al. [9]	FCFS-CI	SPF-CI
KTH-SP2	23 [36]	[44]	50	59
CTC-SP2	1 [37]	[59]	79	85
SDSC-BLUE	38 [47]	[05]	63	74
SDSC-SP2	32 [29]	[15]	66	75

7 Conclusions and Discussion

Scheduling parallel jobs is a hard problem, especially in online contexts. Important information, such as the actual job execution time, is often very imprecise. In particular it is common that users request a much longer runtime than what their jobs actually need, which prevents them from being backfilled and decreases the overall responsiveness of the system. Yet, predicting job execution time from the limited historical information provided by the platform is challenging and often generates only imprecise estimates.

In this work, we showed that a coarse classification of jobs into small and large is sufficient to improve scheduling performance. A simple safeguard mechanism that kills large jobs misclassified as small is important to prevent these jobs from unduly delaying others. Since the misclassification is detected very early, when the job execution time reaches the divider value between classes, which is never more than a few minutes, it results in a small overhead over the average slowdown metrics. We obtained improvements in scheduling performance for all combinations of seven workload traces and four scheduling policies evaluated (see Figure 6). Moreover, in most scenarios, we managed to obtain improvements in scheduling performance comparable to that of clairvoyant schedulers with perfect knowledge of job execution times. Finally, we showed that our performance is not unfair (Figure 8) in the sense that although the performance gain mostly targets small jobs (which are prioritized), it is not detrimental to large jobs.

We claim that in this context, using a classification approach is more effective than using regression for improving scheduling performance. Compared to regression-based techniques, our approach has two major advantages: (i) a two-class classification task is easier to learn than regression, requiring less training data, and (ii) misclassification of large jobs as small is detected very quickly during execution, opposed to regression, where underestimates are evident only after the job executed for the entire actual period. To substantiate this claim, we can compare the improvements obtained by our approach with two regression-based approaches: the relatively simple EASY++ [8], which replaces user-provided runtimes estimates by the average runtime of the two previous jobs from the same user, and the one proposed by Gaussier *et al.* [9], which relies on more elaborate regression technique using an asymmetrical loss function. Both works used the workload traces from SDSC-BLUE, SDSC-SP2, KTH-SP2, and CTC-SP2 and reported improvements over the base EASY-backfilling with FCFS ordering policy (see Table 7). Although there are a few methodological differences (simulation technique, trace cleanups, etc.) between our evaluations, our classification approach combined with FCFS reduced the cumulative bounded slowdown by 50–79%, compared to 29–47% from EASY++, and 5–59% from Gaussier *et al.*. Relying on SPF instead of FCFS allows decreasing the cumulative bounded slowdown even further (59–85%), with most of the gain provided by the classification mechanism. Finally, our mechanism greatly reduces the performance difference between heuristics (without classification, FCFS is significantly worse than SPF or SAF) without loosing most of the fundamental properties that make FCFS an appealing option: its simplicity in terms of explainability to users and its predictable behavior [13]. Consequently, we believe that using the proposed scheme of job size classification is more appropriate for deployment in real HPC platforms than regression-based approaches.

Since the gains we report are particularly substantial, one may wonder whether further gain can still be expected or not. For most of the traces we studied, not only the learning is very good (Table 5) but there is almost no difference between the performance of our classification-based scheme and the one a fully informed (clairvoyant) approach would give (Figure 9), which means that very little gain may be expected from the learning perspective. It may be the case that some gain could be obtained with more elaborate scheduling heuristics though, which could be evaluated by trying to compute lower bound approximations on performance (e.g., using black box optimization as done in [15]).

Note that a potential improvement perspective can be foreseen by closely inspecting the only trace (METACENTRUM) where the weekly performance of the learning did not seem stable (Figure 4). This trace exhibits particularly irregular job submissions with burst of jobs that lead to sudden jumps in the cumulative bounded slowdown (Figure 6). We suspect that our batched (weekly) learning strategy may not be able to adapt well to such rapidly changing situations. Online monitoring of the classification error may then be a good indication that the situation has evolved and that the learning should quickly be updated.

A final future work related to the interplay between learning and scheduling we envision is the exploitation of the information on the uncertainty provided by the learning algorithm. Currently, we only perform a hard assignment to the small/large class whereas the learning algorithm returns an estimation of the probability of belonging to one class or the other. This information could be exploited, for example to perform a less (or more) aggressive prioritization of small tasks and decrease the number of restarted tasks. More generally, although our work advocates the use of classification in this context, we believe that designing scheduling and backfilling strategies that would be provided with a runtime probability distribution (estimated through learning) instead of a simple duration bound would be particularly interesting.

Acknowledgment

This research was financed in part by the french research agency (Energumen ANR-18-CE25-0008) and was partially supported by the Multi-disciplinary Institute on Artificial

Intelligence MIAI at Grenoble Alpes (ANR-19-P3IA-0003). The authors would like to thank Millian Poquet and Michael Mercier for their help and comments. We thank the contributors of the Parallel Workloads Archive, Victor Hazlewood (SDSC SP2), Travis Earheart and Nancy Wilkins-Diehr (SDSC Blue), Lars Malinowsky (KTH SP2), Dan Dwyer, Steve Hotovy (CTC SP2), Ake Sandgren (HPC2N), and The Czech National Grid Infrastructure (MetaCentrum).

References

[1] "TOP500 Supercomputer Sites," https://www.top500.org/.

[2] F. Xhafa and A. Abraham, "Computational models and heuristic methods for grid scheduling problems," *Future Generation Computer Systems*, vol. 26, no. 4, pp. 608–621, 2010.

[3] D. Ye and G. Zhang, "On-line scheduling of parallel jobs in a list," *Journal of scheduling*, vol. 10, no. 6, pp. 407–413, 2007.

[4] P. Brucker, *Scheduling Algorithms*, 5th ed. Springer, 2007.

[5] W. Tang, Z. Lan, N. Desai, and D. 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.

[6] S. Srinivasan, R. Kettimuthu, V. 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.

[7] A. W. Mu'alem and D. G. Feitelson, "Utilization, predictability, workloads, and user runtime estimates in scheduling the IBM SP2 with backfilling," *IEEE Transactions on Parallel and Distributed Systems*, vol. 12, no. 6, pp. 529–543, 2001.

[8] D. Tsafrir, Y. Etsion, and D. G. Feitelson, "Backfilling using system-generated predictions rather than user runtime estimates," *IEEE Trans. Parallel Distrib. Syst.*, vol. 18, no. 6, pp. 789–803, Jun. 2007.

[9] E. Gaussier, D. Glesser, V. Reis, and D. 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*, ser. SC '15. New York, NY, USA: ACM, 2015.

[10] J. Guo, A. Nomura, R. Barton, H. Zhang, and S. Matsuoka, "Machine learning predictions for underestimation of job runtime on HPC system," in *Supercomputing Frontiers*, R. Yokota and W. Wu, Eds. Cham: Springer International Publishing, 2018, pp. 179–198.

[11] D. Carastan-Santos and R. Y. de Camargo, "Obtaining dynamic scheduling policies with simulation and machine learning," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '17. New York, NY, USA: ACM, 2017.

[12] M. Kuchnik, J. W. Park, C. Cranor, E. Moore, N. DeBardeleben, and G. Amvrosiadis, "This is why ML-driven cluster scheduling remains widely impractical," Carnegie Mellon University, Parallel Data Laboratory, Tech. Rep. CMU-PDL-19-103, 2019. [Online]. Available: https://www.pdl.cmu.edu/PDL-FTP/CloudComputing/CMU-PDL-19-103.pdf

[13] D. Carastan-Santos, R. Y. D. Camargo, D. Trystram, and S. Zrigui, "One can only gain by replacing EASY Backfilling: A simple scheduling policies case study," in *CCGrid 2019 - International Symposium in Cluster, Cloud, and Grid Computing*. Larnaca, Cyprus: IEEE, May 2019, pp. 1–10.

[14] J. Lelong, V. Reis, and D. Trystram, "Tuning EASY-Backfilling Queues," in *21st Workshop on Job Scheduling Strategies for Parallel Processing*, ser. 31st IEEE International Parallel & Distributed Processing Symposium, Orlando, United States, May 2017.

[15] A. Legrand, D. Trystram, and S. Zrigui, "Adapting Batch Scheduling to Workload Characteristics: What can we expect From Online Learning?" in *33rd IEEE International Parallel & Distributed Processing Symposium*. rio de janeiro, Brazil: IEEE, May 2019, pp. 1–10.

[16] D. Zotkin and P. J. Keleher, "Job-length estimation and performance in backfilling schedulers," in *Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computing*, ser. HPDC '99. Washington, DC, USA: IEEE Computer Society, 1999.

[17] L. Sant'ana, D. Carastan-Santos, D. Cordeiro, and R. De Camargo, "Real-time scheduling policy selection from queue and machine states," in *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, 2019, pp. 381–390.

[18] D. G. Feitelson, D. Tsafrir, and D. Krakov, "Experience with using the parallel workloads archive," *Journal of Parallel and Distributed Computing*, vol. 74, no. 10, pp. 2967–2982, 2014.

[19] U. Lublin and D. G. Feitelson, "The workload on parallel supercomputers: Modeling the characteristics of rigid jobs," *J. Parallel Distrib. Comput.*, vol. 63, no. 11, pp. 1105–1122, Nov. 2003.

[20] D. G. Feitelson and L. Rudolph, "Metrics and benchmarking for parallel job scheduling," in *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer, 1998, pp. 1–24.

[21] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu, "A density-based algorithm for discovering clusters in large spatial databases with noise," in *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, ser. KDD'96. AAAI Press, 1996, pp. 226–231.

[22] A. P. Dempster, N. M. Laird, and D. B. Rubin, "Maximum likelihood from incomplete data via the EM algorithm," *journal of the royal statistical society, series B*, vol. 39, no. 1, pp. 1–38, 1977.

[23] L. Breiman, "Random forests," *Mach. Learn.*, vol. 45, no. 1, pp. 5–32, Oct. 2001.

[24] S. Nembrini, I. R. König, and M. N. Wright, "The revival of the Gini importance?" *Bioinformatics*, vol. 34, no. 21, pp. 3711–3718, 05 2018. [Online]. Available: https://doi.org/10.1093/bioinformatics/bty373

[25] P. Wei, Z. Lu, and J. Song, "Variable importance analysis: A comprehensive review," *Reliability Engineering and System Safety*, vol. 142, no. C, pp. 399–432, 2015.

[26] E. Gaussier, J. Lelong, V. Reis, and D. Trystram, "Online tuning of EASY-backfilling using queue reordering policies," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 10, pp. 2304–2316, Oct. 2018.

[27] V. C. Stodden, F. Leisch, and R. D. Peng, *Implementing Reproducible Research*, V. Stodden, F. Leisch, and R. D. Peng, Eds. CRC Press, 2014.

[28] E. Dolstra, E. Visser, and M. de Jonge, "Imposing a memory management discipline on software deployment," in *Proceedings of the 26th International Conference on Software Engineering*, ser. ICSE '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 583–592.

[29] P.-F. Dutot, M. Mercier, M. Poquet, and O. Richard, "Batsim: A realistic language-independent resources and jobs management systems simulator," in *Job Scheduling Strategies for Parallel Processing*, N. Desai and W. Cirne, Eds., 2017, pp. 178–197.

[30] H. Casanova, A. Giersch, A. Legrand, M. Quinson, and F. Suter, "Versatile, scalable, and accurate simulation of distributed applications and platforms," *Journal of Parallel and Distributed Computing*, vol. 74, no. 10, pp. 2899–2917, Jun. 2014.

[31] W. Tang, N. Desai, D. Buettner, and Z. Lan, "Analyzing and adjusting user runtime estimates to improve job scheduling on the blue gene/P," in *Proceedings of the 24th IEEE International Parallel and Distributed Processing Symposium*, Apr. 2010, pp. 1–11.

Salah Zrigui is a postdoctoral researcher at the Computer Science Research Institute of Toulouse, France. His interests include machine learning, HPC performance evaluation, and statistical analysis. He is currently working on Analyzing and categorizing energy profiles of HPC applications. He obtained his Ph.D. in computer science From the University of Grenoble and his M.S. in data science from The Grenoble Institute of Technology (Grenoble INP).

Raphael Camargo is an Associate Professor and Vice-Director of the Center of Mathematics, Computing, and Cognition at Federal University of ABC, Brazil. His research interests include high-performance computing, scheduling, and machine learning. He obtained his M.S. and Ph.D. at the University of São Paulo, Brazil, in 2003 and 2007. He published about 45 technical papers in peer-reviewed international journals, conferences, and book chapters and was a member of the program committee from CLUSTER and EuroPar Conferences.

Arnaud Legrand is a senior CNRS researcher at University Grenoble-Alpes since 2004. His research interests encompass the study of large scale distributed systems, theoretical tools (scheduling, combinatorial optimization, and game theory), and performance evaluation, in particular through simulation. He obtained his M.S. and Ph.D. in computer science from the Ecole Normale Supérieure de Lyon, France in 2000 and 2003, and his Habilitation Thesis in 2015 from University Grenoble-Alpes.

Denis Trystram is distinguished professor at the institute of technology of Univ. Grenoble-Alpes. He is an honorary member of the Institut Universitaire de France. He was vice-director of the LIG in Grenoble for 2 years (2014-2015) where he is leading a group on resource management for parallel and distributed systems in a joint team with Inria. Since 2018, he is deputy director of the research dpt. Mathematics/Computer Science at Univ. Grenoble Alpes. He is a well-known specialist of scheduling in parallel and distributed platforms (including HPC clusters, clouds, Internet of Things). Today, he is interested in learning and its link with resource management. He is chairing a research program in Edge Intelligence at the MIAI institute. He served in the program committee of several journals, including Parallel Computing 1996-2020, JPDC 2011-2017 and IEEE TPDS in 2006-2009 and 2014-2016. He published about 100 technical papers in international peer reviewed journals and more than 150 conferences.