Improving the Performance of Batch Schedulers Using Online Job Size Classification
Salah Zrigui, Raphael de Camargo, Denis Trystram, Arnaud Legrand

To cite this version:
Salah Zrigui, Raphael de Camargo, Denis Trystram, Arnaud Legrand. Improving the Performance of Batch Schedulers Using Online Job Size Classification. 2019. hal-02334116

HAL Id: hal-02334116
https://hal.archives-ouvertes.fr/hal-02334116
Preprint submitted on 25 Oct 2019

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 Size Classification

Salah Zrigui¹, Raphael Y. de Camargo², Denis Trystram¹, and Arnaud Legrand¹

¹Université Grenoble Alpes, CNRS, Inria, Grenoble INP, LIG, France
²Federal University of ABC, CMCC, São Paulo, Brazil

Abstract—Job scheduling in high-performance computing platforms is a hard problem that involves uncertainties on both the job arrival process and their execution time. Users typically provide a loose upper bound estimate for job execution times that are hardly useful. Previous studies attempted to improve these estimates using regression techniques. Although these attempts provide reasonable predictions, they require a long period of training data. Furthermore, aiming for perfect prediction may be of limited use for scheduling purposes.

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 fast adaptations to changes in workload characteristics.

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

I. INTRODUCTION

With the ever-increasing demand for computational resources, HPC platforms keep evolving and getting larger and more complex [1], which instigates the need for more adaptive and elaborate scheduling schemes.

One way to deal with such complexity is to develop sophisticated ad-hoc scheduling algorithms that are often too situational, non-generalizable, and hard to reason about. Another, more appealing alternative, is to use a generic scheduling scheme like EASY-backfilling [2] coupled with some index policy. An index policy is a function that considers one or more job characteristics (such as arrival time, processing time, and requested resources), and performs an ordering based on a function of these characteristics. Two notable examples are First Come First Served (FCFS) and Shortest Processing time First (SPF) [3].

One major problem with the aforementioned approach is that many crucial job characteristics, such as job runtimes, are unknown a priori. HPC platforms usually require their users to supply an upper bound to how long a job will execute on the platform. The scheduler also uses this estimate to define which job to include in the backfilling queue and, with policies such as SPF, job ordering in the main execution queue.

Job runtimes estimates are known to be inaccurate at best. Years of practice and experience have shown that users tend to overestimate their job execution times by a large margin, which reduces the effectiveness of backfilling. Considerable research has been conducted to improve the accuracy of these estimates.

Several studies have shown that it is possible to use learning schemes to generate better estimates of job execution times [4]–[7]. However, predicting job execution times using historical data present in workload logs is difficult [8]. These logs usually do not contain important information, such as application name and parameters, data input information, and job structure. Moreover, runtime information such as job placement and machine utilization are available only a posteriori. Finally, existing predictions frequently underestimate job execution times, causing their premature killing after they occupied the platform for a considerable time.

In this work, we propose a different approach. Instead of trying to predict the exact value, we propose a simple classification of the jobs into two classes: small and large. The small class contains all the jobs whose actual runtimes are short, regardless of user-supplied estimates. The large class encompasses all other jobs. We use an online machine learning classification algorithm that exploits user-specific information to learn whether jobs are small or large. We propose a modified version of the EASY algorithm that considers this classification by treating small jobs as high priority jobs.

We perform a thorough evaluation with six full workload traces from HPC platforms and four scheduling policies. We show that:

- Simple binary classification of jobs as small or large is sufficient to improve the performance of all evaluated scheduling policies in all evaluated workload traces;
- The use of a simple safeguard mechanism that kills large jobs misclassified as small enables scheduling improvements similar to those obtainable with a perfect job size classifier;
- Our scheme, combining job size classification and safeguard mechanism, provides improvements close to that obtainable using fully clairvoyant schedulers, with perfect knowledge of actual job execution times.

The remainder of this paper is organized as follows. In Section II we present related studies. In Sections III and IV we discuss the reasoning and the method used for the classi-
fication. Section VII shows the experimental protocol, followed by the experimental evaluation (Section VI). Finally, in Section VII we present the paper conclusions.

II. RELATED WORK

Researchers have used machine learning using two major approaches, which are: (i) improving runtime estimates, and (ii) designing or selecting scheduling policies.

The first approach consists in using machine learning techniques to improve runtime estimates. Feitelson et al. introduced EASY++, a variation of the classical EASY, which replaces user-provided runtimes estimates by the average runtime of the two previous jobs from the same user [4]. 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 [5]. They also showed that prediction could be used more effectively if coupled with a more aggressive backfilling heuristic (namely SPF). But prediction based approaches frequently suffer from the problem of underestimation of runtime times. Guo et al. proposed a framework that can be used to detect runtime underestimates [6], allowing it to adjust job running times.

An interesting phenomenon is that, increasing the inaccuracy (e.g., doubling the user-provided estimates) sometimes improves performance [9]. 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 had a major impact on the effectiveness of accurate predictions, with policies that favor shorter jobs benefiting more. More recent results [5] 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 own studies, we also encountered similar situations (especially when using workload resampling) but this remained an overall statistically insignificant effect.

In a recent study [8], 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 model performances. 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.

Using the second approach, Carastan-Santos and Camargo used synthetic workloads and simulations to determine nonlinear regression functions that improve the slowdown metric [7], generating functions that resemble the Smallest Area First (SAF) policy. Zrigui et al. showed that using a linear combination of job characteristics allowed to build index policies that can significantly improve systems performance [10]. Yet, the authors show that the continuously changing nature of the data makes it very hard to learn online optimal weights for this linear combination and prevents any static policy to be fully effective. Sant’Ana et al. 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 state [11]. These attempts generated promising results but require system administrators to change fundamentally the scheduling policies in existing clusters.

In this work, we propose a simpler approach, which consists in classifying jobs into two classes: small and large. The objective is to allow faster training and adaptations to changes in the workload characteristics, while avoiding other issues of runtime predictions, such as underestimations of runtimes.

III. CHARACTERIZATION OF SMALL AND LARGE JOBS

A. Preliminary definitions

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 systematically studied datasets from six HPC platforms available from the Parallel Workloads Archive [12], and whose main characteristics are shown in Table I.

<table>
<thead>
<tr>
<th>Name</th>
<th>Year</th>
<th># CPUs</th>
<th># Jobs</th>
<th>Duration</th>
</tr>
</thead>
<tbody>
<tr>
<td>HPC2N</td>
<td>2002</td>
<td>240</td>
<td>202,871</td>
<td>42 Months</td>
</tr>
<tr>
<td>SDSC-BLUE</td>
<td>2003</td>
<td>1,152</td>
<td>243,306</td>
<td>32 Months</td>
</tr>
<tr>
<td>SDSC-SP2</td>
<td>1998</td>
<td>128</td>
<td>597,15</td>
<td>24 Months</td>
</tr>
<tr>
<td>CTC-SP2</td>
<td>1997</td>
<td>338</td>
<td>772,222</td>
<td>24 Months</td>
</tr>
<tr>
<td>KTH-SP2</td>
<td>1996</td>
<td>100</td>
<td>284,76</td>
<td>11 Months</td>
</tr>
<tr>
<td>MetaCentrum-zegox</td>
<td>2013</td>
<td>576</td>
<td>795,46</td>
<td>24 Months</td>
</tr>
</tbody>
</table>

In this work, we adopt the simplest model of a 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 estimated runtime \( \hat{p}_j \), provided by the user at job submission and which is an upper bound of \( p_j \leq \hat{p}_j \);
- 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 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 [13]. Yet, the density of estimated runtimes for the six traces shows one or two peaks at small values, showing that most jobs have small processing time estimates (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 estimated and actual runtimes.

But evaluating the actual runtimes of the six traces, we can also notice that they share an interesting similarity, with all distributions having a sharp peak at the small values and a large tail towards longer execution times. This distribution indicates that we could 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.

B. Job classes and their impact

We first examine the characteristics and impact of small jobs in HPC platform usage. To perform this examination, we must first define where the small job class ends, and the large begins. We applied two clustering algorithms, DBSN [14] and EM [15], to divide the classes into two groups. Both generated similar division, shown as a dashed black line in Figure [1]. We also considered a simpler scheme, by using the median of the actual runtimes, represented by the solid green line on Figure [1]. Although the divisions are not the same, we aimed for simplicity and considered that the median is sufficient to separate the initial peak from the rest of the distribution.

We considered a job as small if its runtime was smaller than the median (divider), and consider the job as large otherwise. We can further divide the small job class into two subclasses: (i) non-premature small jobs: short jobs that also had estimated runtimes smaller than divider, and (ii) premature small jobs: short jobs that had estimated runtimes larger than divider.

Table [II] shows the percentage of jobs from each class in the various platforms we tested. We can see that, there is always a significant fraction of premature jobs. Table [III] shows that the total summed runtime and area of these premature jobs occupy a negligible portion of the total runtime and area (less than 0.5%).

Premature small jobs have a wildly over-estimated processing time, causing them to wait longer for execution, which results in large slowdown values. 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.

IV. JOB SIZE CLASSIFICATION

The objective of the job size classifier is to map a set of job features into a class: small or large. We defined the features to use for classification based on two observations: (i) the runtime of a job is highly correlated with the user submission history, and (ii) users often submit more than one job type [8]. One can

\[ a_j = p_j \times q_j \]

1The area \( a_j \) of a job \( j \) is defined as its runtime multiplied by the number of resources it requested: \( a_j = p_j \times q_j \).
We consider that two jobs are in the same category when they have either the same requested processing time, requested number of resources, or submission date. In particular, the runtime of any given job is highly correlated with the runtime of the previous jobs that belong to the same category, as shown in Figure 2 with jobs closer in time having a higher correlation. Consequently, the class (small or large) of the previous jobs that belong to the same category, as shown in Figure 2, with jobs closer in time having a higher correlation.

Based on these observations, we decided to use the following features, shown in Table IV, for classification:

- **Lag features**: contains the class of the previous three jobs from the same category submitted by the user.
- **Aggregation feature**: contains the percentage of jobs of the same category and submitted by the same user that belongs to the small class. This feature aggregates information from older jobs.
- **Temporal features**: The hour of the day, the day of the week, the month, the week, and the quarter in which a job was submitted;
- **Job features**: The estimated execution time and requested number of resources of the job.

We used Random Forests [16] to perform the classifications as they allow to gracefully combine numerical and categorical features. Random forests work by creating multiple decision trees on randomly selected data samples, getting a prediction from each tree, and selecting the best solution by majority voting. Moreover, random forest

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

- **Accuracy** is the ratio of correctly predicted observation over the total number of observations: \( \text{accuracy} = \frac{TN + TS}{T+N+S+P+F+FS} \)
- **Precision** is the ratio of correctly predicted small jobs to the total number of jobs that are predicted to be small: \( \text{precision} = \frac{TS}{TS+FS} \)
- **Recall** is the ratio of correctly predicted small jobs to all observations in the small class: \( \text{recall} = \frac{TS}{TS+FN} \)

Table V shows the results of applying Random Forest with the features presented in Table IV. These are the mean values obtained over all classifications performed during the experimental evaluation (Section VI). For all evaluated traces the value of three indicators was always above 80%.

### Table IV

<table>
<thead>
<tr>
<th>Type</th>
<th>Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Job features</td>
<td>( p_i )</td>
<td>user supplied runtime estimate</td>
</tr>
<tr>
<td></td>
<td>( q_i )</td>
<td>user supplied number of resources</td>
</tr>
<tr>
<td>Temporal</td>
<td>( h )</td>
<td>hour of the day</td>
</tr>
<tr>
<td></td>
<td>( D_{week} )</td>
<td>day of the week</td>
</tr>
<tr>
<td></td>
<td>( d_{month} )</td>
<td>day of the month</td>
</tr>
<tr>
<td></td>
<td>( m )</td>
<td>Month</td>
</tr>
<tr>
<td></td>
<td>( w )</td>
<td>Week</td>
</tr>
<tr>
<td></td>
<td>( Q )</td>
<td>Quarter</td>
</tr>
<tr>
<td>Lag features</td>
<td>( p_{i-1} )</td>
<td>class of the previous job that was submitted by the same user ( i ) and requested equal runtime</td>
</tr>
<tr>
<td></td>
<td>( p_{i-2} )</td>
<td>class of the second to previous job that was submitted by the same user ( i ) and requested equal runtime</td>
</tr>
<tr>
<td></td>
<td>( p_{i-3} )</td>
<td>class of the third to previous job that was submitted by the same user ( i ) and requested equal runtime</td>
</tr>
<tr>
<td></td>
<td>( q_{i-1} )</td>
<td>by the same user ( i ) and requested equal number of resources</td>
</tr>
<tr>
<td></td>
<td>( q_{i-2} )</td>
<td>by the same user ( i ) and requested equal number of resources</td>
</tr>
<tr>
<td></td>
<td>( q_{i-3} )</td>
<td>by the same user ( i ) and requested equal number of resources</td>
</tr>
<tr>
<td></td>
<td>( d_{i-1} )</td>
<td>class of the previous job that was submitted by the same user ( i ) on the same day</td>
</tr>
<tr>
<td></td>
<td>( d_{i-2} )</td>
<td>class of the second to previous job that was submitted by the same user ( i ) on the same day</td>
</tr>
<tr>
<td></td>
<td>( d_{i-3} )</td>
<td>class of the third to previous job that was submitted by the same user ( i ) on the same day</td>
</tr>
<tr>
<td>Aggregation</td>
<td>mean (_\text{ag} )</td>
<td>percentage of jobs that where submitted by the same user ( i ) and requested equal number of resources and belong to the class small</td>
</tr>
<tr>
<td></td>
<td>mean (_\text{gp} )</td>
<td>percentage of jobs that where submitted by the same user ( i ) and requested equal runtime and belong to the class small</td>
</tr>
<tr>
<td></td>
<td>mean (_\text{gd} )</td>
<td>percentage of jobs that where submitted by the same user ( i ) on the same day and belong to the class small</td>
</tr>
</tbody>
</table>

The two types of prediction errors are false large (FL) and false small (FS). False large errors decrease the effectiveness of our approach since it causes small jobs to wait in the queue with large jobs, but causes no harmful effects. False small classifications are a more significant problem since a large job would be executed earlier, causing delay to true small jobs. We prevent this situation by killing false small jobs, i.e., jobs that execute for more than a specified time limit for small jobs.
We tried to be as transparent as possible and to make our work as reproducible as possible [17]. We provide a snapshot of the workflow we used throughout this work as a link to a git repository [18] which includes a nix file that describes all the software dependencies and an R notebook that allows regenerating all the figures.

We consider HPC platforms as a collection of homogeneous resources, with jobs arriving at a centralized waiting queue, following the submission described in the workload logs. We implemented all simulations using Batsim [19], a simulator based on SimGrid [20] that allows evaluating the behavior of scheduling algorithms under different conditions. We evaluate our method using the 6 traces presented in Table I.

### V. EVALUATION FRAMEWORK

#### A. Scheduling policies

We considered four scheduling policies:

- **FCFS**: First Come First Served [2] 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 [21] and is given by: \( WFP_j = (\frac{\text{wait}_j}{p_j})^3 + q_j \). This policy attempts to strike a balance between the number requested resources, the estimated 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 estimated Processing time First [3] orders the jobs by the estimated processing time (\( \hat{p}_j \)) given by the user.
- **SAF**: Smallest estimated Area First [22] orders jobs by their estimated area \( \hat{a}_j = \hat{p}_j \times q_j \).

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. Yet, some studies [10], [22] 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 [23]. When the waiting time of a job surpasses the threshold, the scheduler transfers the job to the head of the queue.

We implemented the four aforementioned policies in conjunction with the EASY [2] backfilling heuristic. When requested, the scheduler selects for execution the next job in the queue, ordered by the scheduling policy. The first time, the scheduler reaches a job that cannot start immediately, due to the immediate lack of resources, it makes a reservation for that job. Then, it schedules the next jobs in the queue that can execute to completion without delaying the job with the reservation.

#### B. Learning and scheduling algorithm

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 \( Q_{\text{small}} \) and \( Q \), respectively. In the first week, 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. The training has three steps. First, we extract job information from the execution logs (line 1 from Algorithm 1). Then we determine the *divider* (line 2) using the median of the past job true execution times. Finally, we use the extracted features and job classes to train the classifier (line 3).

```
Algorithm 1: Update classifier
1 dataset = extract_execution_logs()
2 divider = cluster(dataset)
3 TrainRandomForestClassifier(dataset, divider)
```

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 \( Q_{\text{small}} \) independently, according to a chosen policy (FCFS, SAF, SPF, or SQF), and merge them in a single queue \( Q_{\text{total}} \), with the jobs belonging to the small class first. Finally, resource allocation is done using the EASY heuristic, as shown in Algorithm 2.

```
Algorithm 2: Scheduling
1 Input : Queue of large jobs \( Q \)
2 Queue of small jobs \( Q_{\text{small}} \)
3 Scheduling policy Policy (FCFS|WFP|…)
4 \# Schedule the jobs in the final queue using the EASY heuristic
```

The only extra relevant overhead compared to the EASY scheduling policies are the job classification into classes small or large, which takes a few hundred milliseconds, and the classifier updating, which takes longer. The update includes finding the median execution time over the workload log of the previous week and training the classifier using the pairs \( features, jobclass \). The full execution of this procedure (Algorithm 1) takes only a few seconds and occurs only at the end of every week. Moreover, it runs independently from the scheduler, without blocking it.

**False Small Jobs**: Some large jobs can be wrongly classified as small by the classifier (false small jobs). Although the resource manager may allow these jobs to execute until completion, this would delay the execution of true small jobs.
Algorithm 3: Killing False Small jobs

1. $Q = \{\}$ # queue of large jobs
2. $Q_{\text{small}} = \{\}$ # queue of small jobs
3. job_counter = 0 # number of submitted jobs
4. while Running do
   5. # go through all jobs currently running
   6. if job$_j$.class == "small" & job$_j$.runtime > divider then
   7. kill(job$_j$)
   8. $Q_{\text{small}}$.remove(job$_j$) $Q$.add(job$_j$)
   9. end
10. end

We employ the policy of killing these jobs when the execution time reaches the divider value. We consider that jobs are idempotent, which means they can be killed and restarted without changing the final execution outcome. The scheduler periodically goes through the list of running jobs (Algorithm 3). 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.

If jobs are not idempotent, the resource manager can allow them to execute until completion. The difference in the performance of the two approaches is discussed in detail in Section VI-D.

C. Evaluation metric

There exist several cost metrics, and each evaluates the performance of specific aspects of HPC platforms [24]. We use the bounded slowdown (bsld) metric, which represents the ratio between the time a job spent in the system and its running time. The reasoning behind the slowdown metric is that the response time of a job should be proportional to its runtime. Indeed, it would not seem fair to delay equally short and long jobs. Formally, 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 queue, p$_j$ is the actual execution time, and $\tau$ is a constant that prevents short job times from generating large slowdown values. We set $\tau$ to 60 seconds.

In this work, we use focus on the cumulative bounded slowdown, which is computed as the sum (resp. mean) of bsld of all the job from the beginning of the execution until the current time and is updated every time a job arrives.

VI. EXPERIMENTAL RESULTS

In Section IV, 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 end-to-end performance metrics, such as the average bounded slowdown.

A. Overall impact on scheduling performance

We evaluated the evolution of the cumulative bounded slowdown when applying the EASY-backfilling with the 4 scheduling policies (FCFS, WFP, SAF, and SPF). Figure 3
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 different SPF and SAF generated the lowest cumulative slowdown in all platforms. WFP had cumulative values close to SPF and SAF, while FSCS had the worst values by a large margin in all cases. These results are consistent with previous comparisons of scheduling policies [10], [23].

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 six 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 VI-E.

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. But 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.

B. Individual month improvement

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 somewhat equitably distributed throughout the evaluation period.

We investigate the effects of our approach on individual months in Figure 4. Each pair of connected points represents

Figure 5. Impact of using the classification-idempotent scheduler on the average bounded slowdown of small and large jobs, when compared to the base scheduler.
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 substantially degrade
performance, such as in MetaCentrum-zegox/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 had large jumps in the cumulative slowdown,
such as MetaCentrum-zegox and HPC2N.

C. Impact of prioritizing small jobs over large jobs

Our algorithm reduces the overall bounded slowdown by
prioritizing small jobs. One crucial question is: what is the
impact of favoring small jobs over the jobs classified as large?

We compute the average bounded slowdown for the jobs
from each of the two classes (Figure 5). As expected, the
small jobs had the most substantial reductions in the average
slowdowns. The extent of the reduction is dependent on the
platform and policy and is mostly proportional to the
improvements in the cumulative bounded slowdown, shown
in Figure 5. More importantly, there is only a small increase
in the average slowdown of large jobs.

The use of job size classifier results in extensive improve-
ments for small jobs, with little or no impact on large jobs.
Consequently, we argue that there are no perceivable hidden
costs for large jobs when prioritizing small jobs.

D. Impact of removing the job-killing 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
by killing the job when it reaches the job size divider value.
But we cannot apply it for non-idempotent jobs. A question
that arises is: can we still improve 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
six platforms, for three scenarios: (i) base, (ii) classification-
idempotent, where we kill false small jobs, and (iii) classi-
fication, where we use classification without job-killing.

Preventing job-killing reduces the effectiveness of the clas-
sification in almost all scenarios (Figure 6). Scheduling large
jobs ahead of others risk delaying all the jobs that come after in
the waiting queue, especially true small jobs whose slowdown
value can increase very quickly.

We note, however, that classification without job-killing still
managed to improve the total slowdown for most cases, but to
a lesser 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
very small degradations in performance. We conclude that
removing the safeguard mechanism significantly reduces the
effectiveness of our method without rendering it completely
useless.

E. Comparison with clairvoyant

Finally, we evaluate what would be the improvements
obtained by schedulers that would know in advance the actual
execution time of each job. We compared 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 $(\tilde{p}_j)$, processing times,
(ii) class-clairvoyant, where the scheduler is indicated which
class the jobs belong to (i.e., as if a perfect job class classi-
fication was achieved), and (iii) classification-idempotent, the
method we propose and which only uses estimated execution
times. Although the clairvoyant versions cannot occur in prac-
tice, they provide us with an upper bound on the achievable
improvements.

Using the classification-idempotent results in improvements
comparable to the class-clairvoyant (Figure 7), except for
MetaCentrum-zegox. 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 had 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 in the bounded slowdown metric. It indicates that trying to accurately predict job execution time 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-zegox trace, where there are 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 3), 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 7 we can see that even with full knowledge no significant improvements where made. 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.

VII. Conclusions

Scheduling parallel jobs is a hard problem in general, especially in online contexts, where important information, such as job execution time, is imprecise or missing. Predicting job execution time from the limited information provided by the platform is challenging and often generates only imprecise estimates.

In this work, we show that a more straightforward classification of jobs into small and large classes is sufficient for improving scheduling performance. A simple safeguard mechanism that kills large jobs misclassified as small is important to prevent these jobs from delaying the others. Since the misclassification is detected very early, when the job execution time reaches the divider value between classes, it has a small overhead over the average slowdown metrics. We obtained improvements in scheduling performance for all combinations of six workload traces and four scheduling policies evaluated. Moreover, in most scenarios, we managed to obtain improvements in scheduling performance similar to that of clairvoyant schedulers with perfect knowledge of job execution times.

Furthermore, we can compare the improvements obtained by our approach with two regression-based approaches: EASY++ [4] the one proposed by Gaussier et al. [5]. Indeed, they also used the workload traces from SDSC-BLUE, SDSC-SP2, KTH-SP2, and CTC-SP2 and they report the improvement over the base EASY-backfilling with FCFS ordering policy (see Table VI). Although there are a few methodological differences between our evaluations, our classification approach combined with FCFS reduces the cumulative bounded slowdown by 50-79%, compared to 29-47% from EASY++, and 05-59% from Gaussier et al. Relying on SPF instead of FCFS allows to decrease the cumulative bounded slowdown even further but most of the gain is provided by the classification mechanism.

These results indicate that a classification approach can be 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 estimated period. 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.

Although distinguishing small jobs from large ones proved effective, we believe this classification is too rough. As a future work, we intend to identify other classes of jobs (e.g., long and thin jobs or series of jobs from a given group and whose duration is multi-modal) that could benefit from a specific treatment by the batch scheduler. We also intend to study how the lack of confidence of the classification could be exploited by the scheduling algorithm, similarly to what is done with Bayesian bandits in online learning.

ACKNOWLEDGMENT

This research was financed in part by the French research agency (Energie ANR-18-CE25-0008). The authors would like to thank Millian Poquet for his help and comments and Michael Mercier for his help in the simulation. We thank the contributors of the Parallel Workloads Archive, Victor Hazelwood (SDSC SP2), Travis Earheart and Nancy Wilkins-Diehr (SDSC Blue), Lars Malinowsky (KTH SP2), Dan Dwyer, wood (SDSC SP2), Travis Earheart and Nancy Wilkins-Diehr Michael Mercier for his help in the simulation. We thank the like to thank Millian Poquet for his help and comments and agency (Energumen ANR-18-CE25-0008). The authors would

REFERENCES