An experimental analysis of regression-obtained HPC scheduling heuristics

Lucas de Sousa Rosa, Danilo Carastan-Santos, Alfredo Goldman

A multiple linear regression approach for understanding the trade-offs in learning HPC job scheduling heuristics

Lucas de Sousa Rosa[1] Danilo Carastan-Santos[2] Alfredo Goldman[1]

[1]Institute of Mathematics and Statistics, University of São Paulo
roses.lucas@usp.br, gold@ime.usp.br
[2]Univ. Grenoble Alpes, CNRS, Inria, Grenoble INP, LIG
danilo.carastan-dos-santos@inria.fr

Abstract

Scheduling jobs in High-Performance Computing (HPC) platforms typically involves heuristics consisting of job sorting functions such as First-Come-First-Served or custom (hand-engineered). Linear regression methods are promising for exploiting scheduling data to create simple and transparent heuristics with lesser computational overhead than state-of-the-art learning methods. The drawback is lesser scheduling performance. We experimentally investigated the hypothesis that we can increase the scheduling performance of regression-obtained heuristics by increasing the complexity of the sorting functions. We used multiple linear regression to develop a factory of scheduling heuristics based on scheduling data. This factory uses general polynomials of the jobs' characteristics as templates for the scheduling heuristics. We defined a set of polynomials with increasing complexity between them, and we used our factory to create scheduling heuristics based on these polynomials. We evaluated the performance of the obtained heuristics with simulation experiments considering numerous HPC platform configurations and jobs' characteristics. Our results show that complex polynomials led to unstable scheduling heuristics due to multicollinearity effects in the regression. We also show that a simple linear combination of the jobs' characteristics as a template leads to efficient regression-obtained heuristics. Modifying this linear combination leads to better performances at the price of less transparent heuristics.

1 Introduction

In many fields of science and industry (climate, health, economics, artificial intelligence, *etc.*), High-Performance Computing (HPC) is an essential element to solve complex problems and to process the ever-increasing amount of data being generated. Informally, HPC consists of utilizing highly parallel and distributed computing platforms. We have reached an extreme-scale of such platforms, with ranks such as the Top500 [1] listing supercomputers with millions of interconnected processors.

Among the many important problems that arise at such a scale, resource management is a critical problem that needs to be solved efficiently for a proper use of such platforms. Resource management involves assigning when and where HPC applications (hereafter referred to as jobs) will be processed in an HPC platform. HPC platform users (research groups, companies, *etc.*) submit their jobs to be processed at any given point, and limited information about the jobs is available for decision-making. This decision-making typically requires solving instances of a problem called online parallel job scheduling.

In contrast to many of the theoretical advances, online parallel job scheduling is in practice solved with scheduling heuristics based on waiting queue sorting algorithms, with backfilling algorithms [21] using First-Come-First-Served (FCFS) job ordering being by far the most used. One of possible reasons for this phenomenon is the explainability of the scheduling heuristics, which may be given by their simplicity. Waiting queue sorting heuristics are easy to understand by both the HPC platform maintainers and users, with FCFS ordering arguably being the easiest to understand. Regarding FCFS, this high level of explainability, comes with the drawback of poor scheduling performances [7].

Recent works [6, 17] employ Machine Learning (ML) techniques to exploit simulation and platform workload data to create scheduling heuristics, as an attempt encode scheduling knowledge obtained by ML with explainable and efficient scheduling heuristics. With this regard, regression methods [6] appear as a promising approach, since we can express scheduling knowledge in the form of simple functional forms

of the jobs' characteristics. The hypothesis of a trade-off between simplicity/performance is still present. We may need to sacrifice simplicity of the functional forms to obtain better scheduling performances.

Our work is a step towards clarifying this hypothesis by concerning the following research question: *By using regression methods to create scheduling heuristics, do we gain scheduling performance by using more complex functional forms?*

This work builds upon the work of Carastan-Santos and de Camargo [6] to develop a factory of scheduling heuristics based on scheduling data. This factory uses general polynomials of the jobs' characteristics as templates for the scheduling heuristics. We elaborated an experimental methodology to evaluate polynomials as scheduling heuristics, with increasing levels of complexity.

We found a surprising result that multicollinearity – a phenomenon intrinsic to regression methods – may drastically degrade the scheduling performance of obtained scheduling heuristics constituted by complex polynomials. A functional form constituted by a linear combination of the jobs' characteristics – the simplest of all polynomials evaluated – presented the best scheduling performances. We show examples that we do not need to sacrifice simplicity to obtain efficient scheduling heuristics. More specifically, in this paper we present the following contributions:

- We performed an experimental study on how to obtain online parallel job scheduling policies in the form of polynomials of the jobs' characteristics. We adopted simulation and regression methods to evaluate a class of polynomials with increasing degrees of complexity. Our results indicate that multicollinearity effects may happen when using regression methods with high degree polynomials, leading to instability of the coefficients and inefficient scheduling performances.
- We showed experimental evidence that if we use regression to adjust a function constituted by a linear combination of the jobs' characteristics, we can obtain efficient scheduling policies that can achieve similar performances to state-of-the-art ML-obtained policies.

We organized the remaining of this paper as follows: in Section 2, we present an overview of related works, and Section 3 presents some preliminary definitions of online parallel job scheduling. We present our methodology in Section 4 and our experimental results in Section 5. Lastly, in Section 6, we present our concluding remarks and future works. The code base for this paper is openly abailable at a Gitlab repository[1].

2 Related Work

2.0.1 Scheduling in theory

The research community studied the parallel job scheduling problem under a general problem called multiple-strip packing problem [4]. Given a set of rectangles (jobs) and set of strips (set of computing resources), the objective is to find a packing of the rectangles into the strips, that minimize the height among the used strips. Such problem in the single-strip case is already NP-hard [4]. Many theoretical works [5, 26, 16, 29, 27] proposed approximation algorithms and performance bounds, but many of these works relied on analytically tractable objective metrics such as makespan (i.e., the completion time of the last finishing job). Makespan is arguably less relevant in the online case, when compared to metrics such as waiting time or slowdown (see Section 4), since we do not have information about a "last job" in an online scheduling configuration.

2.0.2 Scheduling in practice: scheduling heuristics

In online parallel job scheduling, many works explore list scheduling [22] based algorithms that rely on waiting queue ordering heuristics. These heuristics can be created by hand-engineering methods [24], and by tuning methods [18, 17], that forecast possible outcomes of a certain heuristic.

It is long known [12], however, that most practitioners employ First-Come-First-Served (FCFS) based heuristics with a backfilling [21] mechanism, or arbitrary job prioritization, using for instance multiple queue priorities [23]. A possible explanation for the popularity of FCFS-based heuristics is the fact that FCFS is (i) easily understandable by both practitioners and users and (ii) it offers desirable properties such as no-starvation (i.e., every job will eventually execute).

[1] *Link omitted due to double-blind*

2.0.3 Machine learning in scheduling

Machine Learning (ML) is used in the context of online job scheduling mainly in two scenarios: (i) to improve scheduling by predicting the jobs' characteristics [14, 19, 30], and (ii) to create novel heuristics by using techniques such as non-linear regression [6] and evolutionary [17] strategies. More recently, deep reinforcement learning methods [10, 28] are being explored to perform online job scheduling.

Our work situates in the ML methods to create novel scheduling heuristics, more specifically in producing knowledge about regression methods to obtain clear and interpretable heuristics. Our work differs from the works of Legrand *et al.* [17] because we want to investigate the simplicity/performance trade-off of regression based methods. The work of Carastan-Santos and de Camargo [6] performed a first step on using regression methods to obtain clear and interpretable scheduling heuristics. Their ML-obtained heuristics, however, relied on functional forms based on terms such as square roots and logarithms, which are hard to interpret in the scheduling sense. We complement the work performed by Carastan-Santos and de Camargo [6] by using a functional form generation procedure to create heuristics in the form of simple polynomials of the jobs' characteristics, taking into account increasing complexity levels.

3 Preliminary Definitions

3.1 Online Parallel Job Scheduling

A typical HPC platform configuration consists by users submitting applications at any moment in time to be processed in the platform. Both the applications and the platform resources are managed by a resources and jobs management system [15] (RJMS).

Among other tasks, the RJMS has to deal with a scheduling problem called online parallel job scheduling problem. As an overview, given a set of n machines (processors) $\mathcal{M} = \{m_1, \ldots, m_n\}$ that need to process parallel jobs $j_1, j_2, \ldots$ that arrive at any time, the online parallel job scheduling problem consists of deciding when and at which machines the jobs will be processed, in a way to maximize a certain performance objective. We assume that these machines are homogeneous (i.e., have the same processing power), and are connected by any interconnection topology.

Under the view of the RJMS, the processors of an HPC platform are modeled as the set of machines $\mathcal{M}$, and the applications submitted by the users are modeled as the jobs j. The RJMS has limited information about the jobs, and this information is only available after the job's submission. The main information available are notably (i) the estimated job's processing time ($\tilde{p}_j$), normally informed by the user, (ii) the number of processors required to process the job (q_j), and (iii) the time when the job was submitted in the platform (r_j, also known as release time).

Another important piece of information that is only available after finishing a job j is its actual processing time (p_j). The estimation of the processing time $\tilde{p}_j$ usually works as an upper bound of the actual processing time p_j. We also consider that the number of processors q_j is fixed and can not change over time (also referred to as rigid jobs). The RJMS registers information about the jobs in a log file, often shared using the *Standard Workload Format* (SWF)[13].

3.2 Scheduling policies and Backfilling

The order to which the jobs are submitted for processing is often defined by a *scheduling policy*, which is a central part of many online parallel job scheduling heuristics. A scheduling policy typically consists by a job sorting function $f(j)$, that takes as input the characteristics of a job j (e.g., $\tilde{p}_j$, q_j, and r_j), and outputs a number that quantifies the priority of processing j. Typically, the lower the value of $f(j)$, the more priority a job j has. The RJMS applies $f(j)$ to all jobs in the waiting queue to set their execution priorities in two events: (i) when a new job arrives in the waiting queue and (ii) when one or more processors are released and becomes available.

On top of a scheduling policy, often a *backfilling* [21] mechanism is applied. Informally, backfilling mechanisms allow a job j to be selected for processing earlier – and breaking the priority order defined by the scheduling policy – if j does not delay the processing of jobs with higher priority. Backfilling is known to consistently bring positive performance effects, regardless of the scheduling policy [7], and it is widely used by practitioners, in conjunction with the First-Come-First-Served (FCFS, $f(j) = r_j$) scheduling policy.

3.3 Scheduling performance metric

In this work we adopt the average bounded slowdown (*AVGbsld*) as the scheduling performance metric. Given a job j, its bounded slowdown (bsld) value can be computed as follows

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

where w_j is the time that a job waited for processing since its submission and τ is a constant that is typically set in the order of 10 seconds, that prevents small jobs from having excessively large slowdown values. The average bounded slowdown takes into account the slowdown average for a set of jobs J and is defined as

$$\text{AVGbsld}(J) = \frac{1}{|J|}\sum_{j\in J}\max\left\{\frac{w_j + p_j}{\max(p_j, \tau)}, 1\right\} \tag{2}$$

AVGbsld is an interesting metric in the sense that it can express the expectation that the waiting time of the jobs should be proportional of their processing time [11], though this metric is more sensible to small jobs (i.e., jobs with small p and q values). *AVGbsld* is also a hard objective to be treated theoretically in online parallel job scheduling, and it is in practice mainly manageable by heuristics.

4 Methodology

We adapted the method of Carastan-Santos and de Camargo [6] to suit to our study. Our method consists of a two-step approach, (i) we gain and register scheduling knowledge by exploiting fast simulations of a *proxy scheduling problem* (i.e., a similar though simpler scheduling problem, see Section 4.1). The strategy is that with simulations and a proxy problem we can perform a larger search of possible scheduling decisions in feasible time. Then (ii) we feed the knowledge observed in the simulations in a linear regression method, that models the observed scheduling knowledge into functions of the jobs' characteristics. The main hypothesis is that these functions will perform as good scheduling policies for the original scheduling problem (see Section 3).

4.1 Simulation step

The key differences of the proxy scheduling problem and the main problem (see Section 3) is that in the proxy problem we (i) have perfect information about the jobs' processing time (i.e., $\tilde{p}_j = p_j$), (ii) it's an off-line scheduling problem (i.e., all information about the jobs (p, q, and r) are known in advance), and (iii) the number of arrived jobs is finite. The idea is to define a similar scheduling problem as the target one, though simpler in such a way that facilitates fast simulations.

Let S and Q two sets of jobs. We frame a proxy scheduling problem as the scheduling of the jobs in Q, in an HPC platform with m interconnected processors where the jobs in S are currently being processed. This configuration is a way to model the waiting queue (Q) and the initial state of the platform (S). The job sets S and Q are generated in the following way: from a large enough job log file (see Section 3, also referred to as trace) N, we randomly select a subtrace $M \subset N$ with size $|S| + |Q|$. The first $|S|$ jobs from M belongs S and the remaining $|Q|$ jobs from M belongs Q.

For a pair of sets of jobs (S, Q), we start the simulation by submitting the jobs from the set S in FCFS order. When the last job of S is submitted, jobs from S may still be processing in the platform, thus representing a possible initial state. At this point we start branching the simulation, with different branch (also referred in the text as *trials*) representing a possible way to schedule the jobs in Q.

Next, we randomly sample permutations Q^* of the set Q, and each sample Q^* represents a branch of simulation. For each sample Q^*, we simulate the scheduling of the jobs, following the order that the jobs are present in Q^*. The total number of permutations of Q increases exponentially in function of the size of Q, and the number of sample permutations to decently represent the permutation space rapidly increases as well. One of the reasons to use an aforementioned scheduling proxy task is that the above simulation branching strategy can only be done in feasible time with fixed (and small, see Section 5) number of jobs in Q.

With $\mathcal{P}$ being the set containing all sampled permutations, when the scheduling simulation of all branches end, we compute and assign a *score* for each job $j \in Q$:

$$score(j) = \frac{\sum_{Q_j^*\in\mathcal{P}(j_0=j)}\text{AVGbsld}(Q_j^*)}{\sum_{Q_k^*\in\mathcal{P}}\text{AVGbsld}(Q_k^*)} \tag{3}$$

The *score* represents the consequence of scheduling a job $j \in Q$ first (represented by j_0 in Equation 3), in terms of the average bounded slowdown of all jobs in Q. Jobs with low *score* resulted in a positive consequence on the average slowdown when they are chosen to be executed first. The set $\{score(j) \mid \forall j \in Q\}$ defines a *score distribution* of the jobs of Q given the initial state S.

The $score(j)$ along with the characteristics p_j, q_j and r_j are the central output of the simulations. We conjecture that with an initial state represented by the scheduling of the jobs in S, sorting the jobs in Q in increasing order of $score(j)$ results in an efficient schedule regarding the $AVGbsld(Q)$ (Equation 2).

We repeat the aforementioned simulation strategy with many samples of job set pairs (S, Q). The idea is that each distinct pair (S, Q) represents a certain scheduling situation, and the computed $\{score(j) \mid \forall j \in Q\}$ represents a good scheduling strategy for this scheduling situation.

4.2 Multiple linear regression step

At the end of the simulation strategy, we have a data-set with the information about the jobs' characteristics p_j, q_j and r_j, and their computed $score(j)$. We hereafter refer to this data-set as *training data-set*. The objective of this next step is to exploit this data-set to obtain a general representation of the values of $score(j)$ in function of the job's characteristics p_j, q_j and r_j.

Let J be all jobs present in the training data-set with their compute *score* values, the problem consists in finding functions $f(p_j, q_j, r_j)$ that provide a good approximation to the *score* values of all jobs $j \in J$. For this task, we modified the original strategy proposed by Carastan-Santos and de Camargo [6]. We defined a function family $\mathcal{F}$, with parametrized functions of the form

$$f(\theta, \mathbf{x}) = \theta^T \mathbf{x} \tag{4}$$

where θ is a parameter vector, and $\mathbf{x}$ is a vector of functional forms of the jobs' characteristics p, q and r.

We defined a function family containing four functions whose vectors $\mathbf{x}$ are presented by Table 1. Each function works as a template for obtaining the scheduling heuristics. Each element in the vectors (also referred to as basis functions) are polynomials of the jobs' characteristics, with degrees in the set $\{1, 2, 3, 4\}$. The reasons behind the choice of these functions are the following:

- We wanted to avoid possible non-linearity between the parameters, which allows us to use multiple linear regression, in which a least squares minimization algorithm can provide optimal parameters θ with regards to the sum of squared loss (see Equation 5).
- We wanted to address the efficiency of using multiple regression to obtain scheduling policies, with functions with increasing levels of complexity. We therefore defined a small set of polynomials with significant complexity increase between them.
- We wanted to avoid less interpretable terms in the scheduling sense, when compared to the terms used in [6] such as square roots and logarithms. We also wanted to avoid polynomials of the jobs' characteristics with unconventional degrees (e.g., 0.5 (square root), -0.5 (inverse square root), 1.5, etc.), and combinations of the jobs characteristics that do not correlate to an already known scheduling heuristic (e.g, derivative functional forms of pr or qr).
- We wanted to evaluate the effects of derivative functional forms of p, q and r and the area (see explanation below) pq, with increasing degrees of penalization for their values.

We defined four functions, Lin, Sq, Cub and Qua. The function Lin is just a linear combination of the jobs' characteristics p, q and r. The others Sq, Cub and Qua are functions that progressively increase the degree of the basis functions, and with multiplicative factors related to pq, which is often referred in the literature as the area of the jobs. We considered derivative factors that correspond to well-known scheduling policies, notably the First-Come First-Served ($FCFS = r$), Shortest Processing Time first ($SPT = p$), Shortest Number of Processors First ($SQF = q$), and Shortest Area First ($SAF = pq$). Therefore, we discarded derivative forms of pr and qr.

We employ a weighted multiple linear regression [8] procedure, which minimizes the weighted sum of squared loss function:

$$\Sigma_{wL} = \sum_{j \in J} \left[(p_j q_j) \cdot (f(\theta, \mathbf{x}) - score(j)) \right]^2 \tag{5}$$

Table 1: Functional forms (templates) of the four parametrized functions used in multiple linear regression

Function Name	Vector $\mathbf{x}$
Lin	$1, p, q, r$
Sq	$1, p, q, r$ p^2, q^2, r^2, pq
Cub	$1, p, q, r$ p^2, q^2, r^2, pq p^3, q^3, r^3 $p^2q, pq^2, (pq)^2$
Qua	$1, p, q, r$ p^2, q^2, r^2, pq p^3, q^3, r^3 $p^2q, pq^2, (pq)^2$ p^4, q^4, r^4 $p^3q, pq^3, (pq)^3$

The weight $(p_j q_j)$ emphasizes that the approximation must perform a good estimation of the *score* of large area jobs (i.e., jobs with p and q large), as they can end up blocking the execution of small jobs, degrading the overall scheduling performance.

After obtaining the coefficients $\hat{\theta}^f$ from the multiple linear regression approximation for all functions $f(\theta, \mathbf{x}) \in \mathcal{F}$, we can measure the approximation quality through the *Mean Absolute Error* function (MAE, Equation 6).

$$MAE(f) = \frac{1}{|J|} \sum_{j \in J} \| f(\hat{\theta}^{\mathbf{f}}, \mathbf{x}) - score(j) \| \tag{6}$$

With these approximated functions we perform a simulation experimental campaign to address how efficient these functions are when used as scheduling policies in an online parallel scheduling scenario.

5 Experimental Results

In this section we present the main results obtained by our work. We produced a training data-set with the $score(j)$ values, and we evaluated the learned scheduling policies by using Amazon's Elastic Compute Cloud (Amazon EC2) `c5a.2xlarge` instance, which offers the second generation AMD EPYC 7002 processor with frequencies up to 3.3 GHz, 8 virtual CPU cores and 16GB of RAM. All simulations (to obtain the training data-set and to evaluate the scheduling policies) were performed with the SimGrid [9] simulation software. To construct the training data-set we defined the size of the sets S and Q as 16 and 32, respectively. From both sets, the jobs' characteristics are obtained from a trace generated with the Lublin and Feitelson [20] workload model.

During the construction of the training data-set we had first to determine how many trials we can perform with a pair (S, Q) in the `c5a.2xlarge` instance, to keep a balance between the precision of calculating the $score(j)$ and the execution time. Figure 1 shows the normalized standard deviation of calculating $score(j)$ in function of the number of trials (which was adapted from [6]), and we added their respective processing times, when running for one pair (S, Q) in the `c5a.2xlarge` instance. We decided to keep 256 thousand trials, as it showed to provide accurate calculations of $score(j)$ and its processing time was acceptable within our budget. Moreover, we used the SciPy's `curve_fit` [25] method to perform the multiple linear regression.

Throughout this section, we refer to the term *on-line scheduling experiment* as being multiple simulations of the on-line scheduling of jobs, whose information are obtained from a certain workload log (trace) or model. For each simulation we choose policies from Tables 1 and 5, and we simulate the on-line scheduling of jobs, using the chosen policies as scheduling heuristic. The output of the on-line scheduling experiment is the average bounded slowdown (see Equation 2) from each simulation performed, using all chosen policies. Each simulation considers a job workload consisting of a fifteen-days jobs submission. When using real workload logs, we assured that there was no overlap between sequences used on distinct on-line scheduling experiments.

Table 2: Functions used in the nonlinear regression and their *Mean Absolute Error*.

Function	$MAE(f)$
$score(j) = Lin$	$4.48 \cdot 10^{-3}$
$score(j) = Sq$	$4.66 \cdot 10^{-3}$
$score(j) = Cub$	$1.05 \cdot 10^{-2}$
$score(j) = Qua$	$6.42 \cdot 10^{-3}$

The simulation of the on-line scheduling works as follows: tasks arrive in a centralized waiting queue and the scheduler performs a reschedule – using a scheduling policy – of the tasks present in this queue in two distinct events: (i) when a task arrives in the queue or (ii) when a resource (set of processors) is released and becomes available. When a job j is selected for processing and if the requested number of processors q_j is lower than the total number of processors available, then q_j processors are reserved for this task and they become unavailable. These processors will become available again only when p_j units of time have passed since the start of the execution of j. If there is not enough processors to process j, then the scheduler waits for one of the two rescheduling events mentioned above. The scheduler has no information about future job arrivals and the jobs' data p, $\tilde{p}$, q, and r are only known to the scheduler when the jobs arrive.

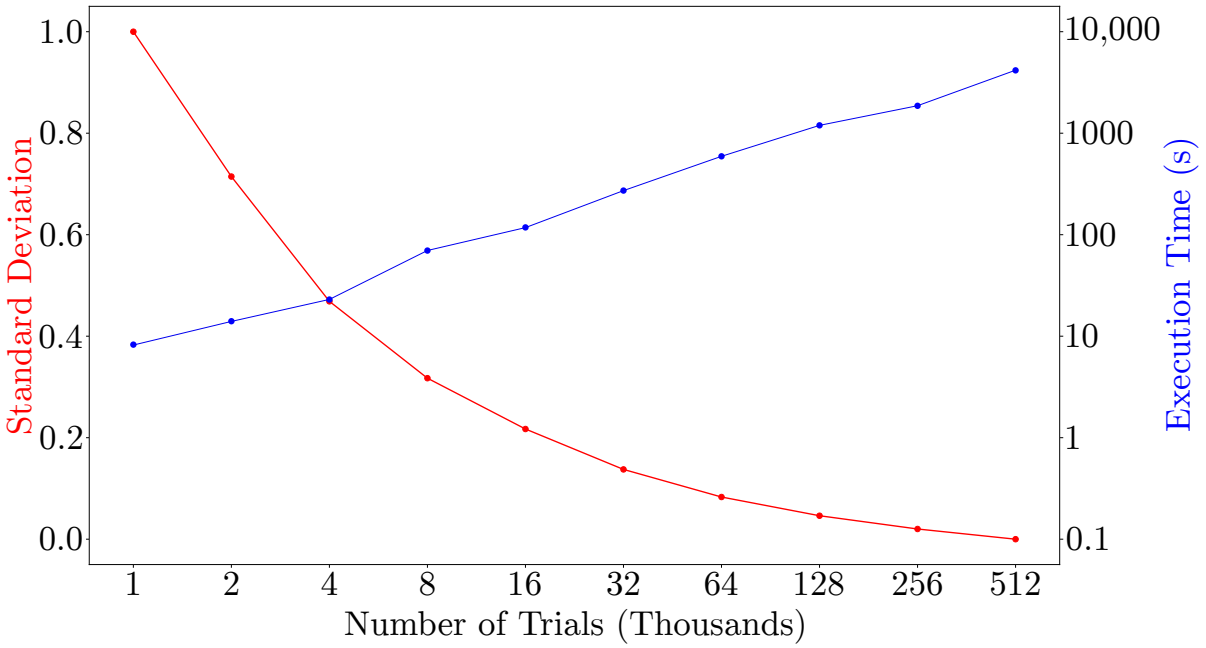


Figure 1: In red, normalized standard deviations for the trial score distributions obtained with different numbers of trials. In blue, the execution time in the chosen instance in logarithmic scale. Figure adapted from [6].

We explored the methodology presented in Section 4 to bring light to several research questions that we elaborate in the following sections.

5.1 Does adding more complex functional forms result in better ML-obtained scheduling policies?

To answer this question, we executed the methodology presented in Section 4 to adjust the functions *Lin*, *Sq*, *Cub* and *Qua* as candidate scheduling policies. The data-set used in the multiple linear regression consists of 7168 jobs and their calculated $score(j)$ values. We considered a synthetic homogeneous platform constituted by 256 processors. The full list of coefficients θ obtained by regression approximation is illustrated in Table 3. We elaborate a qualitative description of these coefficients below.

When comparing the coefficients θ, increasing the degree of the basis functions led to smaller coefficients (close to the order of 10^{-24}) for the functional forms with higher degrees. The functional forms

Table 3: Functional forms of the four parametrized functions used in multiple linear regression and their adjusted coefficients

Function Name	Vector $\mathbf{x}$	Coefficients θ
Lin	$1, p, q, r$	$3.16 \cdot 10^{-2},\ 1.24 \cdot 10^{-7},\ 3.10 \cdot 10^{-5},\ -1.62 \cdot 10^{-7}$
Sq	$1, p, q, r$	$3.86 \cdot 10^{-2},\ 1.33 \cdot 10^{-7},\ -2.95 \cdot 10^{-5},\ -3.63 \cdot 10^{-7}$
	p^2, q^2, r^2, pq	$-2.10 \cdot 10^{-12},\ 7.65 \cdot 10^{-8},\ 2.64 \cdot 10^{-12},\ 9.98 \cdot 10^{-10}$
Cub	$1, p, q, r$	$3.02 \cdot 10^{-2},\ 3.92 \cdot 10^{-7},\ 1.37 \cdot 10^{-4},\ -5.72 \cdot 10^{-7}$
	p^2, q^2, r^2, pq	$-5.03 \cdot 10^{-12},\ -1.04 \cdot 10^{-6},\ 7.44 \cdot 10^{-12},\ 4.49 \cdot 10^{-9}$
	p^3, q^3, r^3	$4.17 \cdot 10^{-17},\ 2.76 \cdot 10^{-9},\ -3.07 \cdot 10^{-17}$
	$p^2q, pq^2, (pq)^2$	$-1.29 \cdot 10^{-13},\ -2.72 \cdot 10^{-11},\ 7.64 \cdot 10^{-16}$
Qua	$1, p, q, r$	$4.58 \cdot 10^{-2},\ -2.08 \cdot 10^{-7},\ -2.64 \cdot 10^{-4},\ -7.56 \cdot 10^{-7}$
	p^2, q^2, r^2, pq	$-1.43 \cdot 10^{-11},\ 1.65 \cdot 10^{-6},\ 1.45 \cdot 10^{-11},\ 2.20 \cdot 10^{-8}$
	p^3, q^3, r^3	$2.56 \cdot 10^{-16},\ 8.72 \cdot 10^{-10},\ -1.26 \cdot 10^{-16}$
	$p^2q, pq^2, (pq)^2$	$5.57 \cdot 10^{-14}, 1.81 \cdot 10^{-10},\ 1.27 \cdot 10^{-16}$
	p^4, q^4, r^4	$-9.20 \cdot 10^{-22},\ -1.33 \cdot 10^{-11},\ 4.15 \cdot 10^{-22}$
	$p^3q, pq^3, (pq)^3$	$-1.75 \cdot 10^{-18},\ 3.71 \cdot 10^{-13},\ 2.17 \cdot 10^{-23}$

Table 4: Variance Inflation Factor (VIF) of the models used in the regression.

Function Name	Vector $\mathbf{x}$	VIF
Lin	p, q, r	1.328, 1.288, 1.216
Sq	p, q, r	3.7534, 9.384, 5.046
	p^2, q^2, r^2, pq	2.520, 8.143, 4.029, 3.765
Cub	p, q, r	12.333, 37.313, 19.957
	p^2, q^2, r^2, pq	45.999, 267.416, 72.771, 125.382
	p^3, q^3, r^3	26.051, 152.388, 31.638
	$p^2q, pq^2, (pq)^2$	67.858, 174.528, 88.216
Qua	p, q, r	40.817, 104.123, 56.525
	p^2, q^2, r^2, pq	430.451, 2841.919, 623.116, 544.329
	p^3, q^3, r^3	1301.604, 10583.406, 1693.613
	$p^2q, pq^2, (pq)^2$	863.788, 2148.213, 1365.561
	p^4, q^4, r^4	448.834, 3585.264, 528.992
	$p^3q, pq^3, (pq)^3$	109.862, 519.405, 176.759

showed instabilities in the coefficients between the functions Lin, Sq, Cub and Qua. We observe this instability by the change in the sign (e.g., from positive to negative) of the coefficients of a specific functional form between the functions Lin, Sq, Cub and Qua. The sign change, in turn, carries a meaning for the schedule. For instance, while Lin prioritizes jobs with small q, since Lin has a positive coefficient for q, function Sq prioritizes jobs with large q, since Sq has a negative coefficient for q.

For Lin, the positive coefficient for q is likely to bring a positive effect in the schedule, since it relates to a known scheduling policy named smaller number of processors first (SQF), which is known [7] to bring better scheduling performances than FCFS. For Sq, the negative coefficient for q is likely to bring negative effects in the scheduling, since it relates to a policy called largest number of processors first (LQF) which is known to bring worse scheduling performances than FCFS, and are often discarded in scheduling policies works [7, 17].

To illustrate these effects, Figure 2 depicts the normalized $score(j)$ values of the functions Lin, Sq, Cub and Qua when varying the values of p and q, and with a fixed value of r. Darker regions in this figure illustrate higher job priorities. We can observe that all functions have different behavior. Lin prioritizes jobs with lower p and q values – which is similar to the quite efficient [7] Shortest Area First (SAF) scheduling policy – Sq, Cub and Qua give more or less priority in specific (less interpretable) regions.

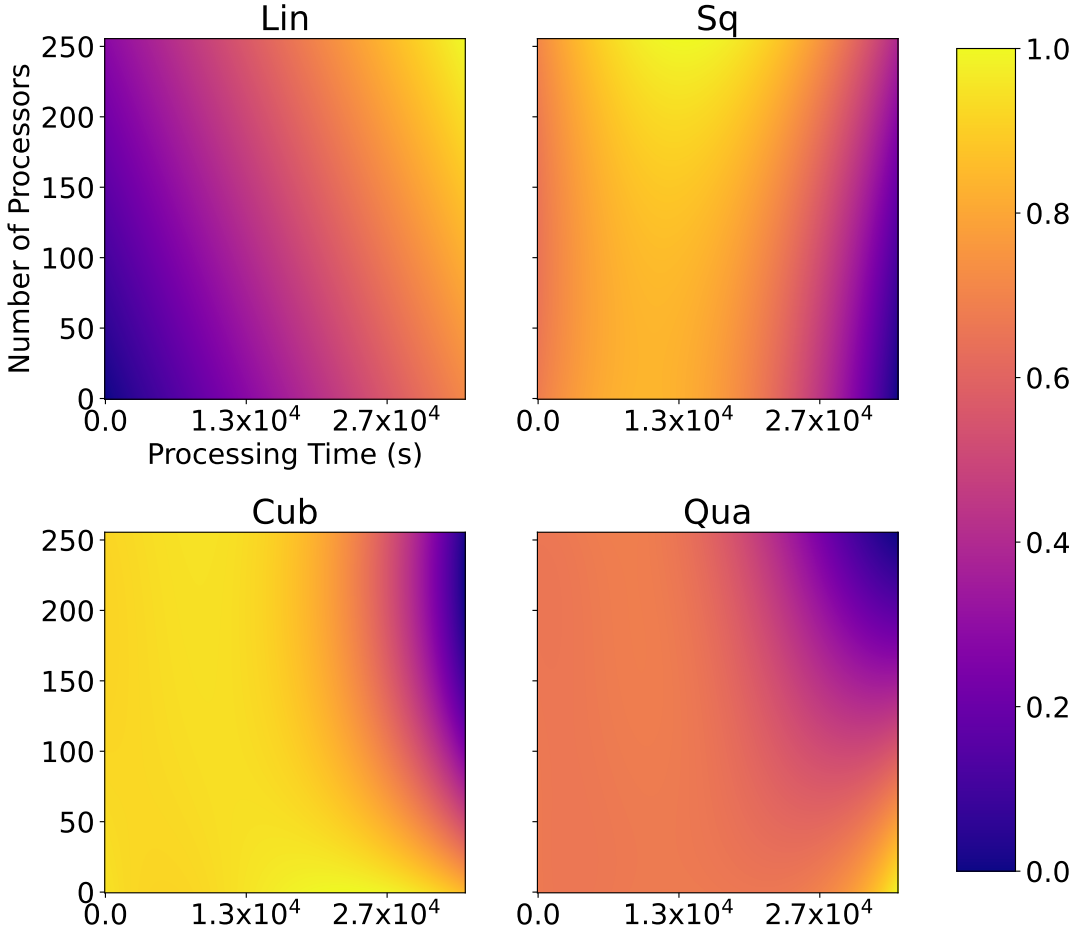


Figure 2: Dependency on the jobs characteristics p and q for the functions Lin, Sq, Cub and Qua.

We evaluated the scheduling performance of the four candidate policies with online job scheduling simulations. In this experiment, the jobs were generated from the Lubin & Feitelson's synthetic workload model (the same used in the simulation step of the methodology, see Section 4.1) and considering an HPC platform constituted by 256 processors.

Figure 3 shows the average bounded slowdown for fifty dynamic scheduling experiments with the above configuration. The results show that the Sq, Cub and Qua functions – which are more complex functions – actually did not perform as well as scheduling policies, presenting the worst results. Although the MAE values remained very close for almost all functions (as shown in Table 2), the performance

of Sq, Cub and Qua showed to be as inefficient as FCFS, thus reinforcing the negative effects of the coefficients for Sq, Cub and Qua. The Lin function showed better results, being comparable to SAF, which is known to be quite efficient. We elaborate on the reasons why Sq, Cub and Qua are inefficient in the next section.

5.1.1 Multicollinearity results in poor scheduling policies

This instability in the coefficients of Sq, Cub and Qua indicates that these functions suffer from the phenomenon of multicollinearity [3], that happens when the functional forms are highly correlated between each other. At this light, multicollinearity may be happening when we add derivative functional forms in conjunction to the jobs' characteristics p, q and r (e.g., q^2, q^3, q^4).

To verify this multicollinearity hypothesis, we calculated the Variance Inflation Factor [2] (VIF) for the functional forms present in the functions Lin, Sq, Cub and Qua. Variance inflation factors range from 1 upwards. The numerical value for VIF tells (in decimal form) what percentage the variance is inflated for each coefficient. For example, a VIF of 1.9 tells you that the variance of a particular coefficient is 90% larger than what you would expect if there was no multicollinearity (i.e., if there was no correlation with other functional forms).

Therefore, one way to classify the correlation of the functional forms in terms of their VIF is the following: (i) not correlated ($VIF = 1$), (ii) moderately correlated ($1 \leq VIF \leq 5$), and (iii) highly correlated ($VIF > 5$). The calculated VIF values are described on Table 4. The Sq function displays VIF values greater than 5 as in q, r, q^2. Furthermore, the higher degree functions (Cub and Qua) presented very high values, reaching values greater than 10 thousand as in the term q^3 of the function Qua. We conjecture therefore that adding more derivative functional forms will result to larger VIFs, and the multicollinearity effect will be present.

These results suggest that obtaining new scheduling heuristics for HPC platforms depends on choosing the most important functional forms, keeping the models more simple. In the rest of the experiments we put emphasis in the Lin function, since it was the function with the lowest level of multicollinearity.

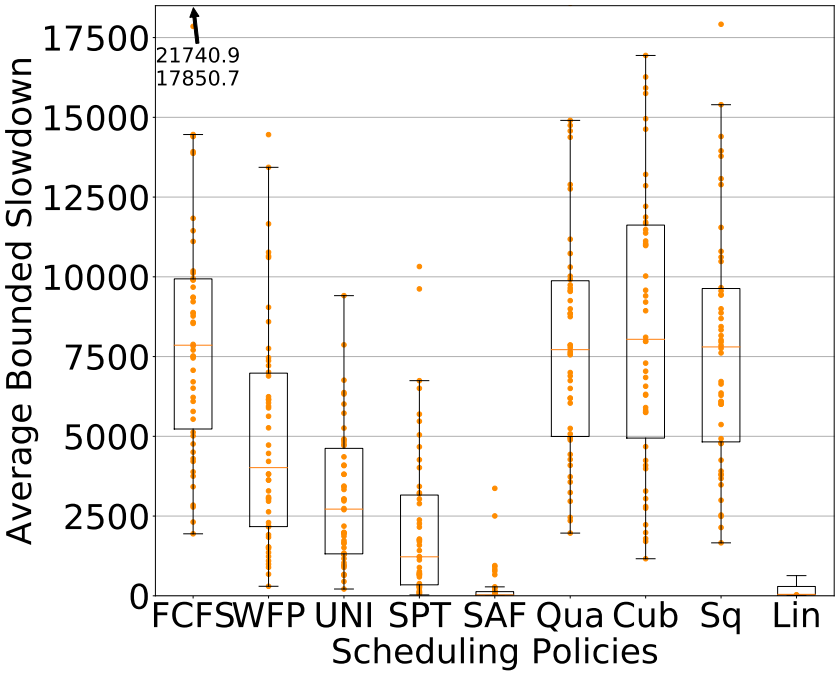


Figure 3: Scheduling performance comparison with jobs generated from Lublin & Feitelson workload model.

5.2 How good a linear combination can be as scheduling heuristic?

The following experiments aimed to measure the ability of the Lin function to generalize for different types of workloads, obtained from real workload traces of large scale HPC platforms, comparing it

Table 5: Scheduling policies used for comparison.

Name	Function
FCFS	$score(j) = r_j$
SPT	$score(j) = \tilde{p}_j$
SAF	$score(j) = \tilde{p}_j \cdot q_j$
WFP3 [24]	$score(j) = -(w_j/\tilde{p}_j)^3 \cdot q_j$
UNICEF [24]	$score(j) = -w_j/(\log_2(q_j) \cdot \tilde{p}_j)$
F2 [6]	$score(j) = \sqrt{\tilde{p}} \cdot q + 2.56 \cdot 10^4 \cdot \log_{10}(r)$

Table 6: Used real workload traces characteristics.

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

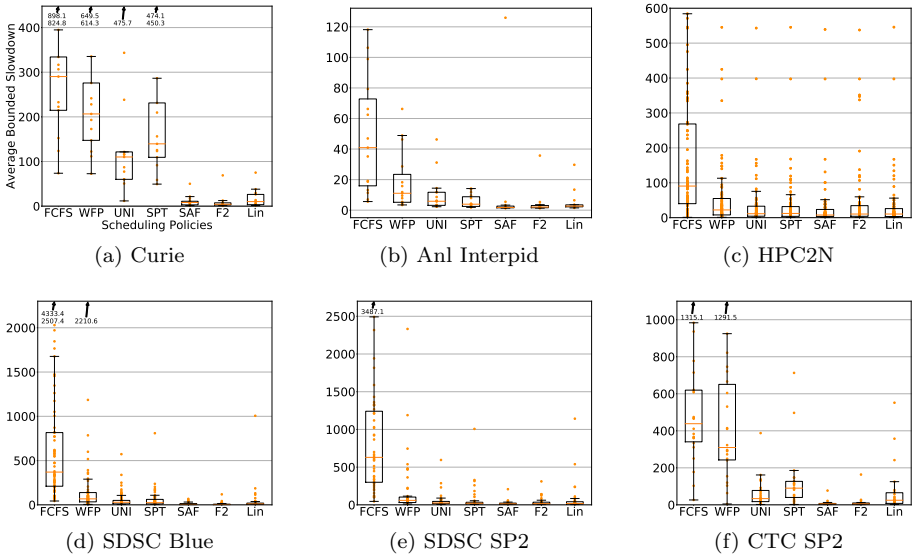
with the policies presented in Table 5. The used traces (see Table 6) are publicly available at Parallel Workloads Archive [13].

We performed on-line scheduling experiments taking into account three situations, (i) scheduling using perfect information about the jobs processing time p, (ii) scheduling using jobs' processing time estimates $\tilde{p}$, and (iii) scheduling using processing time estimates $\tilde{p}$, and in conjunction with aggressive backfilling [21].

The third aforementioned scenario is the one that is arguably closer to a real HPC platform situation. It is important to note that the policies SPT, SAF and *Lin* do not prevent job starvation. The F2 policy prevents starvation, but its anti-starvation capacity diminishes over time due to its term $\log_{10}(r)$. We evaluated these policies without any starvation prevention mechanism. Job starvation can be effectively implemented for any of them by a thresholding mechanism, where a job receives the highest priority if it passes a certain threshold (waiting time, estimated slowdown, *etc.*).

5.2.1 Scheduling experiments using the actual job processing time p

The results of the experiments are illustrated in Figure 4. The linear form presented low average bounded slowdown for the different workloads. In addition, the dispersion of the data was small in almost all cases, ensuring greater stability and predictability. Finally, the *Lin* function showed one of the three best results for all experiments. Even though each workload trace was different, the linear function as a scheduling policy proved to be able to be generalized for different types of tasks.



(a) Curie (b) Anl Interpid (c) HPC2N

(d) SDSC Blue (e) SDSC SP2 (f) CTC SP2

Figure 4: Scheduling performance results from real HPC platform workload logs using actual processing time.

5.2.2 Scheduling experiments using user estimated job processing time $\tilde{p}$

Figure 6 shows similar results, that is, that the *Lin* function continues to present lower average bounded slowndowns for all workload traces. Unlike the previous test, in which the actual processing times were used, when using the estimated processing times $\tilde{p}$, performance degradation is expected for all policies (the estimated values are usually rough and inaccurate estimates).

There were some cases where the average slowndowns were considerably high, this happened due to the presence of uncommon fifteen-day workloads. A clear example is the HPC2N trace that contains bursts of jobs with large processing time, overloading the platform regardless of policy.

5.2.3 Scheduling experiments using estimated processing times and aggressive backfilling

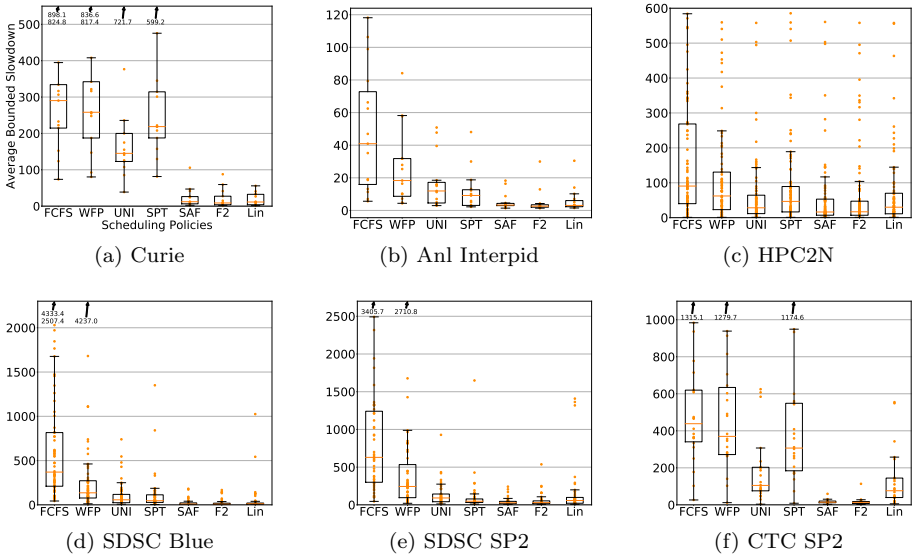
As the most realistic scenario, in addition to the estimated processing time, there was the addition of aggressive backfilling (used to reduce the idleness of the platform). Figure 6 shows that *Lin* did not benefit as much from backfilling as the FCFS policy. Nevertheless, even though the *Lin* function lagged behind the F2 and SAF policies in most cases, it still resulted in lower average bounded slowdowns and/or lower differences between the extreme quartiles for most scenarios, demonstrating good scheduling performances.

6 Conclusions and Future Work

The complexity of online parallel scheduling algorithms has always been a negative factor for their deployment in real scenarios. Scheduling heuristics that sorts the jobs in a waiting queue (scheduling policies) were always the scheduling algorithm of choice in real deployments. This choice is arguably due to the simplicity and interpretability of scheduling heuristics. Machine Learning (ML) methods to create scheduling heuristics seem a compelling approach to providing simple, transparent, and efficient scheduling heuristics. However, an under-explored aspect of such methods is how much we can gain scheduling performance if we increase the complexity of the ML-obtained heuristics.

In this work, we explored a multiple regression method to understand the trade-offs between the transparency/simplicity and the performance of ML-obtained scheduling heuristics. We first created a data set of scheduling observations by performing fast scheduling simulations of numerous scheduling situations. Then we feed this data set in a multiple linear regression method to adjust the functions of the jobs' characteristics as scheduling policies. We created scheduling policies with increasing levels of complexity to understand such trade-offs.

We evaluated the scheduling performance of these ML-obtained scheduling policies using many distinct simulation scenarios. These policies are parametrized functions of the characteristics of the jobs



(a) Curie (b) Anl Interpid (c) HPC2N

(d) SDSC Blue (e) SDSC SP2 (f) CTC SP2

Figure 5: Scheduling performance results from real HPC platform workload logs using user estimated processing times.

(i.e., jobs' processing time p, number of processors q, and arrival time r).

We first showed evidence that the scheduling performance of these policies is sensible to the obtained parameters (coefficients) of the functions. We observed a multicollinearity effect when using multiple linear regression with complex functions constituted by several derivative terms of the job characteristics. This multicollinearity increased the instability in the coefficients of the obtained scheduling policies and resulted in poorly performing scheduling policies.

We had to keep the functions simple to obtain efficient scheduling policies with multiple linear regression. A linear combination (hereafter referred to as *Lin*) of the three terms p, q, and r was the scheduling policy that achieved the best results among the obtained scheduling policies. Other policies containing more derivative terms resulted in instabilities in the obtained parameters and worse scheduling performances. This instability appears quickly as we add derivative terms in the policies. Removing terms to reduce the multicollinearity would result in functions very similar to the *Lin* policy. The *Lin* policy is appealing since it mixes three simple known scheduling policies. The coefficients obtained through multiple linear regression ponder the relevance of each of these three policies. With all of the above in mind, we decided not to mitigate the multicollinearity in this work by, for instance, removing terms in the other policies.

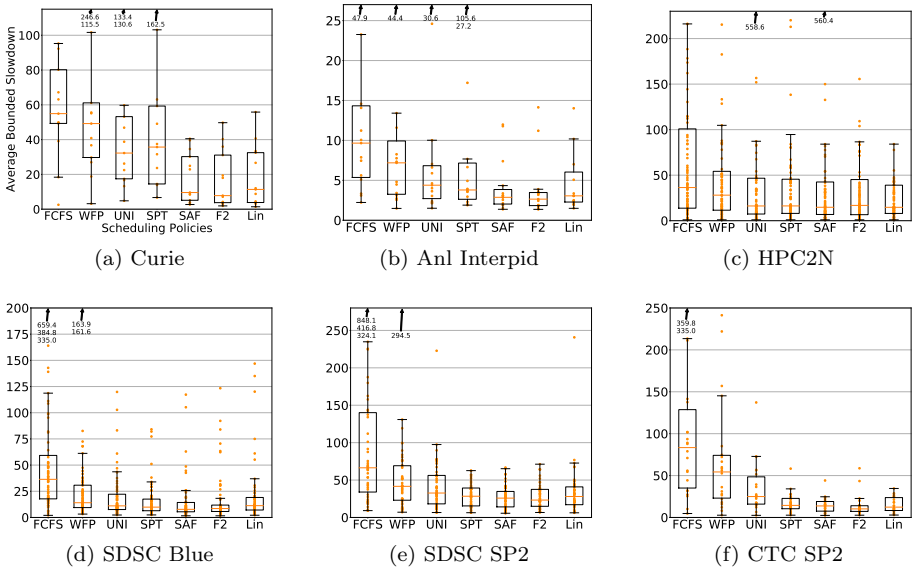
We then performed an extensive experimental campaign to evaluate the scheduling performance of the *Lin* policy. Thanks to the simulations, we evaluated *Lin* considering many distinct real-world situations. Our simulation results show that *Lin* presents good scheduling performances in the majority of the evaluated scenarios, with performances comparable to the quite efficient [7] Shortest Area First (SAF).

We also observed that ML-obtained policies that combine the characteristics p, q, and r in more unusual ways like the policy F2 [6] possibly lead to slightly better performances than *Lin*. This slight performance increase comes with the drawback of lesser interpretability, as it is harder to interpret why the square roots and logarithms in F2 lead to better scheduling. We can obtain better policies than *Lin* by increasing the complexity of combining p, q, and r, but we do not foresee significant performance gains.

For future work, we will improve the proposed method by (i) using platform-related features (e.g., platform utilization, remaining time of the processing jobs, etc.) and external factors (e.g., time of the day) in addition to p, q and r, (ii) by performing a search of possible scheduling heuristics taking into account the level of multicollinearity of the functional forms, and (iii) by using automatic ways, such as symbolic regression, to combine the functional forms.

References

[1] TOP500 Supercomputer Sites. `https://www.top500.org/`, 2018. Online; last access 30 november



(a) Curie (b) Anl Interpid (c) HPC2N

(d) SDSC Blue (e) SDSC SP2 (f) CTC SP2

Figure 6: Scheduling performance results from real HPC platform workload logs using user estimated processing times, with the addition of the aggressive backfilling algorithm.

2018.

[2] Akinwande, M. O., Dikko, H. G., and Samson, A. Variance inflation factor: As a condition for the inclusion of suppressor variable(s) in regression analysis. *Open Journal of Statistics 05* (2015), 754–767.

[3] Alin, A. Multicollinearity. *Wiley Interdisciplinary Reviews: Computational Statistics 2* (5 2010), 370–374.

[4] Baker, B. S., Coffman, Jr, E. G., and Rivest, R. L. Orthogonal packings in two dimensions. *SIAM Journal on computing 9*, 4 (1980), 846–855.

[5] Bougeret, M., Dutot, P., Jansen, K., Otte, C., and Trystram, D. Approximation algorithms for multiple strip packing. In *Approximation and Online Algorithms, 7th International Workshop, WAOA 2009, Copenhagen, Denmark, September 10-11, 2009. Revised Papers* (2009), pp. 37–48.

[6] Carastan-Santos, D., and de Camargo, R. Y. Obtaining dynamic scheduling policies with simulation and machine learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (New York, NY, USA, 2017), SC '17, ACM, pp. 32:1–32:13.

[7] Carastan-Santos, D., De Camargo, R. Y., Trystram, D., and Zrigui, S. One can only gain by replacing easy backfilling: A simple scheduling policies case study. In *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)* (2019), pp. 1–10.

[8] Carroll, R., and Ruppert, D. *Transformation and Weighting in Regression*. Chapman & Hall/CRC Monographs on Statistics & Applied Probability. Taylor & Francis, 1988.

[9] Casanova, H., Giersch, A., Legrand, A., Quinson, M., and Suter, F. Versatile, scalable, and accurate simulation of distributed applications and platforms. *Journal of Parallel and Distributed Computing 74*, 10 (June 2014), 2899–2917.

[10] Fan, Y., Lan, Z., Childers, T., Rich, P., Allcock, W., and Papka, M. E. Deep reinforcement agent for scheduling in hpc. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)* (2021), pp. 807–816.

[11] Feitelson, D. G. Metrics for parallel job scheduling and their convergence. In *Workshop on Job Scheduling Strategies for Parallel Processing* (2001), Springer, pp. 188–205.

[12] Feitelson, D. G., Rudolph, L., Schwiegelshohn, U., Sevcik, K. C., and Wong, P. Theory and practice in parallel job scheduling. In *Workshop on Job Scheduling Strategies for Parallel Processing* (1997), Springer, pp. 1–34.

[13] Feitelson, D. G., Tsafrir, D., and Krakov, D. Experience with using the parallel workloads archive. *Journal of Parallel and Distributed Computing 74*, 10 (2014), 2967–2982.

[14] Gaussier, E., Glesser, D., Reis, V., and Trystram, D. Improving backfilling by using machine learning to predict running times. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (New York, NY, USA, 2015), SC '15, ACM, pp. 64:1–64:10.

[15] Georgiou, Y. *Resource and Job Management in High Performance Computing*. PhD thesis, PhD Thesis, Joseph Fourier University, France, 2010.

[16] Hurink, J. L., and Paulus, J. J. Online algorithm for parallel job scheduling and strip packing. In *International Workshop on Approximation and Online Algorithms* (2007), Springer, pp. 67–74.

[17] Legrand, A., Trystram, D., and Zrigui, S. Adapting batch scheduling to workload characteristics: What can we expect from online learning? In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)* (2019), pp. 686–695.

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

[19] Li, J., Zhang, X., Han, L., Ji, Z., Dong, X., and Hu, C. Okcm: improving parallel task scheduling in high-performance computing systems using online learning. *The Journal of Supercomputing 77*, 6 (2021), 5960–5983.

[20] Lublin, U., and Feitelson, D. G. The workload on parallel supercomputers: Modeling the characteristics of rigid jobs. *J. Parallel Distrib. Comput. 63*, 11 (nov 2003), 1105–1122.

[21] Mu'alem, A. W., and Feitelson, D. G. Utilization, predictability, workloads, and user runtime estimates in scheduling the IBM SP2 with backfilling. *IEEE Transactions on Parallel and Distributed Systems 12*, 6 (2001), 529–543.

[22] Pinedo, M. L. *Scheduling: theory, algorithms, and systems*. Springer, 2016.

[23] Rodrigo, G. P., Östberg, P.-O., Elmroth, E., Antypas, K., Gerber, R., and Ramakrishnan, L. Towards understanding hpc users and systems: a nersc case study. *Journal of Parallel and Distributed Computing 111* (2018), 206–221.

[24] Tang, W., Lan, Z., Desai, N., and Buettner, D. Fault-aware, utility-based job scheduling on BlueGene/P systems. In *Cluster Computing and Workshops, 2009. CLUSTER'09. IEEE International Conference on* (2009), IEEE, pp. 1–10.

[25] Virtanen, P., Gommers, R., Oliphant, T. E., Haberland, M., Reddy, T., Cournapeau, D., Burovski, E., Peterson, P., Weckesser, W., Bright, J., van der Walt, S. J., Brett, M., Wilson, J., Millman, K. J., Mayorov, N., Nelson, A. R. J., Jones, E., Kern, R., Larson, E., Carey, C. J., Polat, İ., Feng, Y., Moore, E. W., VanderPlas, J., Laxalde, D., Perktold, J., Cimrman, R., Henriksen, I., Quintero, E. A., Harris, C. R., Archibald, A. M., Ribeiro, A. H., Pedregosa, F., van Mulbregt, P., and SciPy 1.0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods 17* (2020), 261–272.

[26] Ye, D., Han, X., and Zhang, G. Online multiple-strip packing. *Theoretical Computer Science 412*, 3 (2011), 233 – 239. Combinatorial Optimization and Applications.

[27] Ye, D., and Zhang, G. On-line scheduling of parallel jobs in a list. *Journal of scheduling 10*, 6 (2007), 407–413.

[28] Zhang, D., Dai, D., He, Y., Bao, F. S., and Xie, B. Rlscheduler: An automated hpc batch job scheduler using reinforcement learning. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis* (2020), pp. 1–15.

[29] Zhuk, S. Approximate algorithms to pack rectangles into several strips. *Discrete Mathematics and Applications dma 16*, 1 (2006), 73–85.

[30] Zrigui, S., de Camargo, R. Y., Legrand, A., and Trystram, D. Improving the performance of batch schedulers using online job runtime classification. *Journal of Parallel and Distributed Computing 164* (2022), 83–95.