Obtaining Dynamic Scheduling Policies with Simulation and Machine Learning

Danilo Carastan-Santos, Raphael Yokoingawa de Camargo

Obtaining Dynamic Scheduling Policies with Simulation and Machine Learning

Danilo Carastan-Santos
Univ. Grenoble Alpes, CNRS, Inria, LIG
Grenoble, France 38000
Universidade Federal do ABC
5001 Avenida dos Estados
Santo André, Brazil 09210-580
danilo.santos@ufabc.edu.br

Raphael Y. de Camargo
Universidade Federal do ABC
Santo André, Brazil 09210-580
raphael.camargo@ufabc.edu.br

ABSTRACT

Dynamic scheduling of tasks in large-scale HPC platforms is normally accomplished using ad-hoc heuristics, based on task characteristics, combined with some backfilling strategy. Defining heuristics that work efficiently in different scenarios is a difficult task, specially when considering the large variety of task types and platform architectures. In this work, we present a methodology based on simulation and machine learning to obtain dynamic scheduling policies. Using simulations and a workload generation model, we can determine the characteristics of tasks that lead to a reduction in the mean slowdown of tasks in an execution queue. Modeling these characteristics using a nonlinear function and applying this function to select the next task to execute in a queue dramatically improved the mean task slowdown in synthetic workloads. When applied to real workload traces from highly different machines, these functions still resulted in important performance improvements, attesting the generalization capability of the obtained heuristics.

KEYWORDS

Scheduling, High Performance Computing, Simulation, Machine Learning

1 INTRODUCTION

The on-line scheduling of tasks in large-scale HPC platforms is a notoriously and increasingly complicated subject to be tacked by the scheduling systems. The need to address numerous scheduling factors leads to the development of sophisticated scheduling algorithms that are often difficult to reason about or hard to be deployed in real systems. In this light, an appealing alternative is to use scheduling policies to perform the scheduling. Scheduling policies are functions that take as input the characteristics of the tasks (*e.g.* processing time, requested amount of cores, waiting time, *etc.*) and they output a value that denotes the priority of the task. These scheduling policies are often designed in an *ad-hoc* manner, generally based on intuitions regarding which rules the scheduling policies must impose to achieve a good scheduling performance. One common practice of the scheduler systems is to add a backfilling mechanism in conjunction to the scheduling policy. The backfilling increases the utilization of the HPC platform and consistently improves scheduling performance.

Another common practice of HPC platform maintainers is to register information about the tasks that have been executed in the platform. These workload logs contain several important information regarding the characteristics of the tasks such as the characteristics mentioned above and the processing time estimate of the tasks that was provided by the user. In light of the ever increasing amount of information generated by HPC platforms and the need for simple and efficient scheduling solutions, the main question raised by this work is: *Is it possible to design a simple procedure, that employs simulation and machine learning techniques, to extract general and simple scheduling policies from existing workload logs, which perform better than the current ad-hoc scheduling policies?*

In this work, we present a technique based on simulation and machine learning algorithms to generate simple scheduling policies represented by nonlinear functions. These nonlinear functions effectively model the scheduling behavior of the tasks under several distinct situations and, when used as scheduling policies, performs a good improvement in the global scheduling of these tasks. More specifically, this work presents the following contributions:

(1) We show that it is possible to generate efficient scheduling policies in the form of nonlinear functions, obtained from general workload characteristics, that considerably improves the global scheduling, when compared to classical and state-of-the-art *ad-hoc* scheduling policies;
(2) We propose a simple simulation procedure and a machine learning strategy, based on nonlinear regression, to observe the effects of scheduling decisions over tasks obtained from a workload model over distinct conditions, and to model these effects into nonlinear functions that can be used as on-line scheduling policies;
(3) We show that these obtained scheduling policies perform impressively well in scheduling tasks from the same workload model used to observe the scheduling effects, with performances up to 14 times better than the best performing *ad-hoc* scheduling performance in the most realistic

SC'17, Denver, Colorado USA

setting evaluated (*i.e.* in conjunction with a backfilling algorithm and considering the user estimates of task processing times to perform scheduling decisions);

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

The remainder of this paper is organized as follows: In Section 2 we present the closely related works and in Section 3 we present the proposed strategy to obtain on-line scheduling policies. We present the main results (*i.e.* the obtained scheduling policies and its scheduling performances) in Section 4, and the conclusions in Section 5.

2 RELATED WORK

Over the history of scheduling study, many works attempted to tackle the problem of scheduling with a wide range of approaches, covering from integer linear programming [3, 11] to genetic algorithms [13, 20] and neural networks [1, 2]. Xhafa and Abraham [24] present a review of recent computational models and heuristics for scheduling in HPC platforms. While these works show good improvements in scheduling, one aspect that these works share in common is that they are either computationally complex or hard to be deployed in real scenarios.

With regard to simpler scheduling policies, most of the policies are defined by *ad-hoc* intuitions about how the scheduling should behave in order to achieve good performance, from the classical policies with simple reasoning such as First-Come, First-Served (FCFS), Shortest Processing Time First (SPT) [19] and Longest Processing Time (LPT) [19] to the smarter and more complex reasoning policies such as WFP3 [22] and UNICEF [22].

One aspect of these *ad-hoc* scheduling policies is that they perform well only for some workload characteristics and HPC platform configurations. The main difference from our work is that we use observations over scheduling decisions results, using simulations with several distinct workloads, to determine what are the best combinations of task characteristics that result in improvements over the average bounded slowdown of all tasks in the queue. The objective is to obtain policies that generalize better over different workloads and HPC platforms.

A noticeable phenomenon is the divergence between theory and practice in the problem of scheduling tasks in HPC platforms [9]. With the introduction of the backfilling algorithms [16] – whose reasoning is to allow a task with low priority to be executed before a higher priority task if this low priority task does not delay the higher priority one – in practice most recent systems responsible for scheduling tasks in HPC platforms implement either FCFS with aggressive backfilling (called EASY [16] algorithm) or the same aggressive backfilling with some scheduling policy to sort the waiting tasks. For example, SLURM job management system, a well known scheduling system for HPC platforms [25], uses the EASY algorithm or, alternatively, a multi-factor scheduling algorithm, which is the aggressive backfilling with a scheduling policy defined as a linear combination of priority factors (waiting time, size, share factors, *etc.*). The coefficients of this linear function – whose values establish the correlation between these priority factors – are defined by the HPC platform maintainer. Other job management systems such as TORQUE [21], PBSpro [17] or MOAB [6] implement similar approaches with their own specificities. Georgiou [12] presents a detailed review of these job management systems. The common point of these schedulers is that they use simple heuristics for scheduling, that work reasonably well in a variety of situations. Our work targets this kind of system, with the differential that, instead of *ad-hoc* heuristics, we define a procedural way to find simple and efficient scheduling policies for the type of tasks normally submitted for execution on these systems.

3 FINDING SCHEDULING POLICIES WITH SIMULATION AND MACHINE LEARNING

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

3.1 Scheduling Background

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

- The estimated processing time e_t of the task informed by the user;
- The actual processing time r_t of the task (only known after the task has been executed);
- The resource requirement of the task, measured as the number of cores n_t;
- The arrival time s_t of the task (also called release date).

Although some data sets have additional information, the selected variables are available in most real workload traces, shared using the Standard Workload Format (SWF) [10].

The performance of scheduling algorithms are mostly evaluated [8] using the *bounded slowdown* objective function which is defined as follows for a task t:

$$bsld = \max\left(\frac{w_t + r_t}{\max(r_t, \tau)}, 1\right) \quad (1)$$

where w_t is the time that task t waited for execution (*i.e.* the time that t starts its execution minus the arrival time s_t) and τ is a constant, with a typical value of 10s, that prevents small tasks from having excessively large slowdown values. Similarly, we can define the *average bounded slowdown* which is the slowdown average over a sequence of tasks T:

$$AVEbsld(T) = \frac{1}{|T|} \sum_{t \in T} \max\left(\frac{w_t + r_t}{\max(r_t, \tau)}, 1\right) \quad (2)$$

In this work, all scheduling evaluations are performed using this objective function.

3.2 Simulation Scheme

We considered for simulation an HPC platform represented by an homogeneous cluster compounded by n_{max} resources (cores). We defined two sets of tasks to be executed, S and Q. Set S contains $|S|$ tasks that are executed in any order at the beginning of the simulation. Tasks from Q, which are used to extract information about scheduling performance, start to arrive after all tasks from S arrived. This scheme provides a realistic way to represent an initial resource state of the cluster before the arrival of tasks from Q.

The first step is to determine, using simulations, how the scheduling performance is affected when a task t is selected for execution, under different sets of tasks and resource states. To obtain this information, several tuples of task sets (S, Q) were generated. For each tuple (S, Q), we define $\mathcal{P}$ as a collection of random permutations of Q and $\mathcal{P}(t_0 = t)$ as the subset from all permutations where t is the first task in the permutation. We then simulate the scheduling execution for each pair (S, p) for all $p \in \mathcal{P}$. We call these pairs (S, p) as *trials* of the tuple (S, Q). On each trial, each task $t \in Q$ is submitted for execution in the order as they appear in p. We then assign a *score* for each task $t \in Q$:

$$score(t) = \frac{\sum_{p_j \in \mathcal{P}(t_0 = t)} AVEbsld(p_j)}{\sum_{p_k \in \mathcal{P}} AVEbsld(p_k)} \quad (3)$$

The score denotes the impact of assigning a task $t \in Q$ to execute first, in the average bounded slowdown of all tasks in Q. The set of scores for all tasks $t \in Q$ constitute a *trial score distribution* of the tasks of Q under initial resource state S. Typical distributions, shown in Figure 1, contains most scores slightly above or below the mean $\frac{1}{|Q|} = \frac{1}{32} = 0.031$. Tasks with lower scores have a more positive impact in the average bounded slowdown when they are chosen to be executed first.

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

3.3 Machine Learning Scheme

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

Table 1: Base functions chosen for nonlinear regression.

Name	Description
`id`	`id(x)` = x
`log`	`log(x)` = $\log_{10} x$
`sqrt`	`sqrt(x)` = $\sqrt{x}$
`inv`	`inv(x)` = $1/x$

Let T_r be the set of all tasks from all sets Q generated in the simulation phase (see Section 3.2). For a task $t \in T_r$, we have a 4-tuple $(r_t, n_t, s_t, score(r_t, n_t, s_t))$ obtained from the previously computed trial score distributions. This 4-tuple denotes the *observation* of the scheduling performance behavior of the task t. Given a collection $\mathcal{F}$ of nonlinear functions, the problem consists in finding the function $f(r, n, s) \in \mathcal{F}$ that better fits the distribution $score(r, n, s)$ generated from all tasks $t \in T_r$.

The functions in $\mathcal{F}$ are functions of the form $f = (c_1\alpha(r))\ op_1\ (c_2\beta(n))\ op_2\ (c_3\gamma(s))$. We call α, β and γ as *base functions* and they can be any of the functions presented in Table 1. Operators op_1 and op_2 are any of the operators sum (+), multiplication ($\cdot$) or division ($\div$). Coefficients c_1, c_2 and c_3 denotes the relative importance of the base functions and are obtained by a nonlinear regression fitting. We chose the base functions presented in Table 1 because they, in conjunction, can express a wide variety of nonlinear relationships, while maintaining a tangible amount of functions to perform the fit. We employ a weighted nonlinear regression [4] procedure, which minimizes the error:

$$error = \sum_{t \in T_r} ((r_t n_t) \cdot (f(r_t, n_t, s_t) - score(r_t, n_t, s_t)))^2 \quad (4)$$

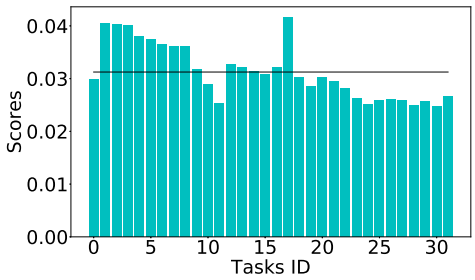
We used the weight $(r_t n_t)$ to emphasize that the fit must perform a good estimation of the score of bigger tasks (*i.e.* tasks with large r and n values). This is based on the argument that tasks that consume a large amount of resources for a long period of time have a potential of blocking the execution of many smaller tasks, degrading the overall scheduling performance.

Once we perform the fit of all functions $f \in \mathcal{F}$, we use a *rank* function (Equation 5) to evaluate the overall fitness of each nonlinear function f to estimate the $score(r_t, n_t, s_t)$ for all tasks $t \in Tr$.

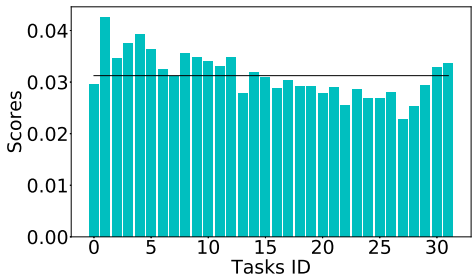
$$rank(f) = \frac{1}{|Tr|} \sum_{t \in T_r} \|f(r_t, n_t, s_t) - score(r_t, n_t, s_t)\| \quad (5)$$

4 RESULTS

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



(a) One example of a trial score distribution



(b) Another example of a trial score distribution

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

Table 2: Scheduling policies used for comparison.

Name	Function
FCFS	$score(t) = s_t$
SPT	$score(t) = r_t$
WFP3	$score(t) = -(w_t/r_t)^3 \cdot n_t$
UNICEF	$score(t) = -w_t/(\log_2(n_t) \cdot r_t)$

In the simulations to generate the distribution $score(r, n, s)$ we considered an HPC platform compounded by $n_{max} = 256$ homogeneous cores. We used sets of tasks S and Q with $|S| = 16$ and $|Q| = 32$ tasks. All simulations were performed using SimGrid [5].

We compared the performance of our scheduling policies with a selection of classical scheduling and smart *ad-hoc* policies, used in real HPC platforms (Table 2). Two classical and well known policies are First Come First Served (FCFS), where tasks are scheduled by the arrival order, and Shortest Processing Time First (SPT), where tasks with smaller processing times are scheduled first. We also used the WFP3 (WFP, for short) and UNICEF (UNI, for short) policies [22], which are based on the processing time (r_t), requested number of cores (n_t), and waiting time (w_t) of the task. The reasoning of WFP is that shorter and/or older tasks should be largely favored, while preventing the starvation of large tasks. UNI in its turn attempts to provide a fast turnaround for small tasks by favoring them.

4.1 Machine Learning: Obtained Nonlinear Functions

The first step towards obtaining the nonlinear functions for usage as scheduling policies is to produce the distribution $score(r, n, s)$. We start by generating permutations of the set of tasks Q, which are used to construct the trial score distributions. Enumerating and simulating the execution of all permutations of a set of tasks of size $|Q| = 32$ is unfeasible and, therefore, we need to define a suitable number of permutations that generate accurate trial score distributions. For that, we selected one tuple (S, Q) and generated the trial distributions with increasing amount of trials, repeating the simulation procedure ten times per number of trials, and measuring the standard deviation of the estimated scores. Figure 2 shows that the standard deviation drops quickly with increasing amount of trials. With 256 thousand trials, the resulting normalized standard deviation was 0.02. We decided to use 256 thousand trials, since its simulation takes less than 11 minutes using SimGrid [5] on an Intel Xeon E5-2620v2 six-core CPU.

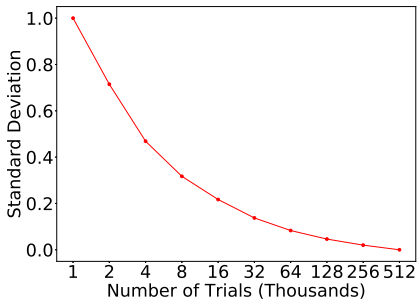


Figure 2: Normalized standard deviations for the trial score distributions obtained with different numbers of the trials.

After obtaining the trial score distributions, we generated the distribution $score(r, n, s)$ and performed the nonlinear regression using the function `leastsq()` from the SciPy [14] Python library. Table 3 shows the four best functions obtained with regard to the Equation 5. These functions were mathematically simplified, with a merging of the coefficients c_1, c_2 and c_3 into a single coefficient $c_3/(c_1 c_2)$, in front of the $log(s)$ term.

A noticeable phenomenon is their similarity, with all functions constituted by a sum of two factors, one containing parameters r and n and another with the dependency on $log(s)$. Considering the large values of the constant before the $log(s)$ term, the functions emphasize largely that tasks that arrived earlier (*i.e.* with lower s values) should be prioritized in order to maintain lower slowdowns (remembering that tasks with lower score value have a high scheduling priority). Figures 3b and 3c illustrate the strong dependency on the submission time for all policies F1 to F4, with tasks that arrive earlier receiving a large priority (darker colors) over more recent tasks.

The second important factor is the size of the task, a product of two functions $f(r)$ and $g(n)$ of the processing time and number of cores used by the tasks. The policies F1 to F4 differ in the relative importance given to each of these values (r and n), with F1

Table 3: The four higher ranked nonlinear functions obtained using nonlinear regression.

ID	Nonlinear Function
F1	$\log_{10}(r) \cdot n + 8.70 \cdot 10^2 \cdot \log_{10}(s)$
F2	$\sqrt{r} \cdot n + 2.56 \cdot 10^4 \cdot \log_{10}(s)$
F3	$r \cdot n + 6.86 \cdot 10^6 \cdot \log_{10}(s)$
F4	$r \cdot \sqrt{n} + 5.30 \cdot 10^5 \cdot \log_{10}(s)$

and F2 imposing a heavier penalization for increasing amounts of requested cores n, F4 penalizing more higher processing times r, and F3 penalizing higher values of r and n equally. Figure 3a shows that, for a fixed value of s, higher priorities are given for tasks that have either required smaller processing times or number of cores.

These results comply with the general intuition – in which tasks with small processing time, small requested amount of cores and that were submitted earlier should be prioritized – that is adopted by most of the *ad-hoc* scheduling policies. The main difference is the adoption of two separate and independent terms, one considering only task size, and the other submission time.

4.2 Scheduling Performance: Workload Model

In this subsection we aim to answer the following question: *Can the nonlinear functions, obtained using the procedure from Section 4.1, perform well as scheduling policies for tasks generated by the Lublin and Feitelson [15] workload model in the following scenarios?*

- Using the actual processing time r in the scheduling decisions and the same number of cores $n_{max} = 256$ from the simulation scheme;
- Using the actual processing time r in the scheduling decisions, but increasing the number of cores to $n_{max} = 1024$;
- Using the processing time estimate e provided by the user, instead of the actual processing time r, to perform the scheduling decisions;
- Using the processing time estimate e provided by the user, but performing the scheduling using the aggressive backfilling algorithm;

We should emphasize that in all scenarios we used the same set of nonlinear functions, obtained using the actual processing time r of tasks and $n_{max} = 256$ cores. Since the functions are parametrized by the number of cores (n), arrival time (s) and processing time r (which can be substituted by the user estimate e), they can be used in different scenarios. The objective was to evaluate the generalization capabilities of the generated nonlinear functions.

For all experiments, we define a *dynamic scheduling experiment* as being the simulation of the execution of ten distinct sequences of tasks from the same workload trace, using the same scheduling policy. Each sequence contains all tasks submissions over a period of fifteen days and we made sure that there was no overlap between the sequences.

The on-line scheduling algorithm 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 cores) is released and becomes available. When a task t is selected for execution and if the requested number of cores n_t is lower than the total amount of cores available, then n_t cores are reserved for this task and they become unavailable. These cores will become available again only when r_t units of time have passed since the start of the execution of t. If there's not enough cores to process t, then the scheduler waits for one of the two rescheduling events mentioned above.

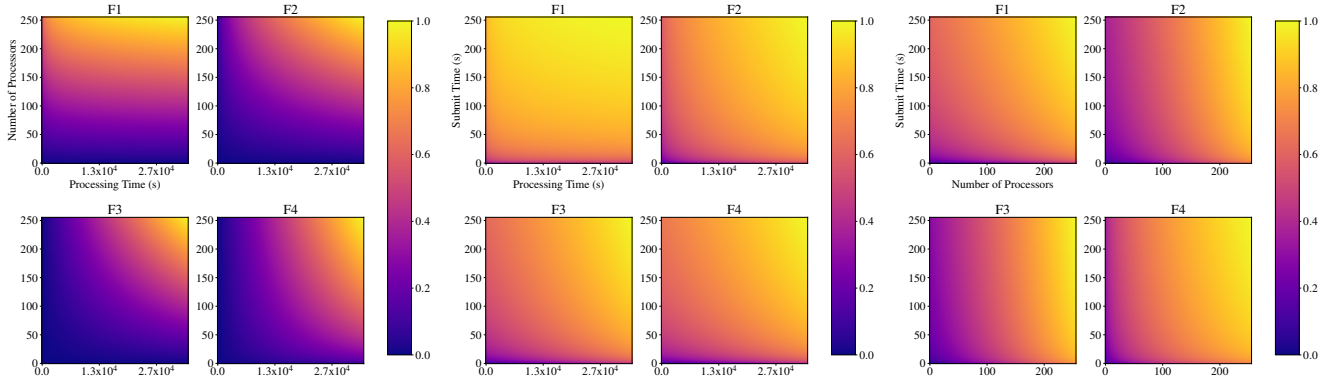
4.2.1 Scheduling using actual task runtimes r. In this experiment we generated a task queue with characteristics from the workload model of Lublin and Feitelson [15] and considering a HPC platform constituted by $n_{max} = 256$ cores, which are the same settings used in the simulations to generate the nonlinear functions. Figure 4a shows the average bounded slowdown for all tasks in the set Q, with the orange line representing the median of the average bounded slowdown, the box limits representing the upper and lower quartiles, and the whiskers representing the lowest and highest values outside the the box limits but still inside the range of 1.5 times the difference between the upper and lower quartiles. The figure shows that all obtained nonlinear functions performed notoriously well as scheduling policies, when compared to the performance of the scheduling policies from Table 2. The nonlinear function F1 provided the best performance, followed by functions F2, F3 and F4. This result was expected, since we used the same configuration from the simulation phase where we captured the scheduling observations. Moreover, the functions F1, F2, F3 and F4 were the four best fitting functions, with decreasing levels of fitness. Therefore, a decrease in the scheduling performance in the same order was also expected.

Figure 4b shows the results of the dynamic scheduling experiments when we considered an HPC platform with $n_{max} = 1024$ cores. The tasks were generated using the Lublin and Feitelson workload model configured for a cluster with 1024 cores, so that tasks sent to the waiting queue would use between 1 and 1024 cores. We can see that, compared to the other scheduling policies, the obtained nonlinear functions continued to perform considerably better, indicating that the obtained scheduling policies have some generalization capability regarding the number of cores in the HPC platform.

4.2.2 Scheduling using user estimated task runtimes. In this experiment, instead of using the processing time r_t in the scheduling decisions, we utilize the *estimated* processing time e_t of the task that is previously provided by the user. The actual processing time r_t in this case is used only to simulate the execution of the task. For the workload model used in this work, we used the user runtime estimate model of Tsafrir *et al.* [23] to generate the processing time estimates.

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

Figures 5a and 5b show the dynamic scheduling experiments results, using estimated processing times e, for HPC platforms constituted by $n_{max} = 256$ and $n_{max} = 1024$ cores. As expected, all



(a) Processing time *versus* amount of cores

(b) Professing time *versus* submit time. Older tasks have small submit time values

(c) Amount of cores *versus* submit time. Older tasks have small submit time values

Figure 3: Dependency on the parameters r, n and s for the four best nonlinear functions obtained.

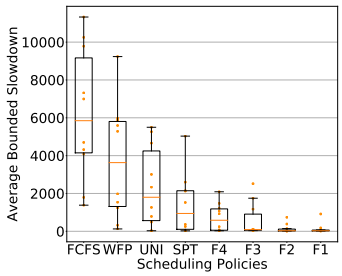
Table 4: Median of the average bounded slowdowns from Subsections 4.2 and 4.3.

Experiment	FCFS	WFP	UNI	SPT	F4	F3	F2	F1
Workload model, $n_{max} = 256$, actual runtimes r	5846.87	3630.66	1799.74	943.59	583.89	89.93	29.65	**29.58**
Workload model, $n_{max} = 1024$, actual runtimes r	10315.62	7759.03	4310.26	4061.44	1518.73	831.18	244.80	**217.13**
Workload model, $n_{max} = 256$, runtime estimates e	5846.87	6021.69	3561.56	4415.27	719.88	405.68	207.05	**33.03**
Workload model, $n_{max} = 1024$, runtime estimates e	10315.62	9713.40	5930.50	7573.58	2605.45	2065.47	1292.64	**249.80**
Workload model, $n_{max} = 256$, aggressive backfilling	842.66	654.81	470.72	623.86	329.49	163.74	45.72	**32.82**
Workload model, $n_{max} = 1024$, aggressive backfilling	3018.94	3792.40	2804.38	3024.49	1571.95	1055.82	490.77	**223.52**
Curie workload trace, actual runtimes r	227.67	182.95	93.76	132.59	20.25	10.66	**3.58**	10.38
Anl Interpid workload trace, actual runtimes r	30.04	11.78	6.03	3.34	1.94	**1.71**	1.87	2.14
SDSC Blue workload trace, actual runtimes r	299.83	44.40	20.37	21.77	14.33	10.38	**4.31**	10.22
CTC SP2 workload trace, actual runtimes r	439.72	309.72	29.87	87.55	19.02	14.06	**5.32**	10.27
Curie workload trace, runtime estimates e	227.67	251.54	135.53	213.03	48.45	24.98	**12.47**	21.85
Anl Interpid workload trace, runtime estimates e	30.04	17.82	11.42	5.44	4.15	3.15	**2.57**	2.64
SDSC Blue workload trace, runtime estimates e	299.83	94.87	39.69	36.42	24.26	10.16	**9.88**	12.14
CTC SP2 workload trace, runtime estimates e	439.72	369.93	98.58	290.39	31.23	21.58	**13.78**	15.14
Curie workload trace, aggressive backfilling	59.03	49.23	24.35	35.72	24.54	23.91	**18.69**	21.73
Anl Interpid workload trace, aggressive backfilling	8.56	6.00	4.01	3.70	3.52	2.87	**2.54**	2.64
SDSC Blue workload trace, aggressive backfilling	36.40	17.76	13.07	10.20	**9.37**	10.18	9.66	11.97
CTC SP2 workload trace, aggressive backfilling	74.96	54.32	24.06	17.32	14.12	14.40	**10.77**	14.07

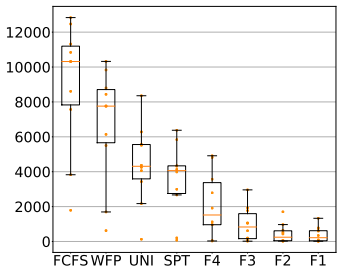
scheduling policies had a considerable performance degradation, except the FCFS, which does not use task processing times. Nevertheless, the median of the average bounded slowdown generated by policies F1, F2, F3 and F4 was between 4.94 and 107.92 times better for the scenario with $n_{max} = 256$ and between 2.27 and 23.74 times better for the scenario with $n_{max} = 1024$, when compared to the best performing *ad-hoc* scheduling policy.

4.2.3 Scheduling using estimated runtimes and agressive backfilling. In this experiment, we used the aggressive backfilling algorithm in conjunction with the scheduling policies. In such setting, when a rescheduling event occurs, the tasks are reordered in the queue using a scheduling policy and then we apply the aggressive backfilling algorithm to check if there's one or more tasks further back in the queue that, if selected for execution, will not delay the first task in the queue. In this case, all the scheduling decisions (the scheduling policy and the backfilling) are made over the requested processing time e_t, with the actual processing time r_t used only to simulate task execution. This setting is the most realistic setting we can elaborate using tasks generated from a workload model.

Figures 6a and 6b show that the introduction of the aggressive backfilling algorithm resulted in a overall increase in performance, with the FCFS policy with backfilling (previously mentioned as the EASY algorithm) taking the most advantage of the backfilling strategy. Although some benefit was achieved, the obtained nonlinear functions had the smallest benifts from the backfilling. This occurs

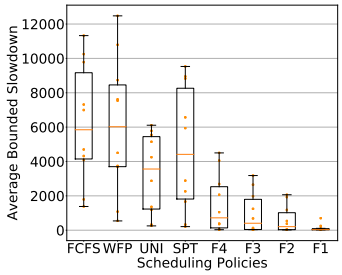


(a) Tasks characteristics generated from the Lublin and Feitelson [15] model and $n_{max} = 256$

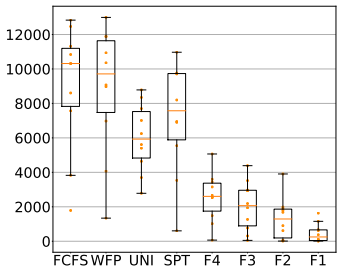


(b) Tasks characteristics generated from the Lublin and Feitelson [15] model and $n_{max} = 1024$

Figure 4: Scheduling performance results with tasks characteristics generated from a workload model and using actual processing time in the scheduling decisions.



(a) Tasks characteristics generated from the Lublin and Feitelson [15] model, with processing time estimates generated from the Tsafrir *et al.* [23] model and $n_{max} = 256$



(b) Tasks characteristics generated from the Lublin and Feitelson [15] model, with processing time estimates generated from the Tsafrir *et al.* [23] model and $n_{max} = 1024$

Figure 5: Scheduling performance results with tasks generated from a workload model and using user estimated processing times in the scheduling decisions.

because the better the initial scheduling, the lower the possibilities task backfilling. Nevertheless, the performance of the obtained nonlinear functions is still significantly better when compared to the other *ad-hoc* scheduling policies. For instance, the median average slowdown for the F1 strategy was more than 12 times smaller than the best *ad-hoc* policy for both 256 and 1024 core machines.

4.3 Scheduling Performance: Real Workload Traces

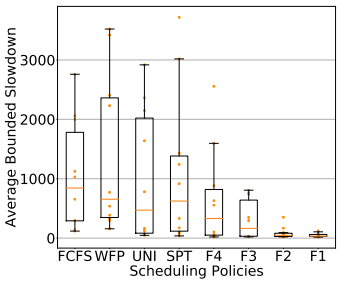
We also evaluated whether the obtained scheduling policies generalize well to highly different workloads types, obtained from real workload traces, and HPC platform configurations. In this light, in this subsection we attempt to answer the following questions:

- Can the obtained scheduling policies perform well when scheduling a set of tasks extracted from real workload traces and executed in a simulated HPC platform similar to the one where the traces were obtained?
- With the same setting from the previous question, but using the user estimated processing times e to perform the scheduling decisions, can the obtained nonlinear functions perform well as scheduling policies?
- Can the obtained nonlinear functions benefit from the aggressive backfilling algorithm and perform well in the scenario from the previous question?

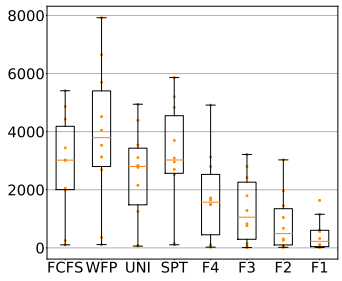
We used the traces described in Table 5, which are publicly available at the Parallel Workloads Archive [10]. To better evaluate the generality of the obtained nonlinear functions, we chose a set of traces from computer HPC configurations ranging from 338 to 163,840 cores, mean utilizations from 59.6% to to 85.2% and measurements dates from year 1997 to 2011.

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

4.3.1 Scheduling using the actual task runtimes r. Figure 7 shows the dynamic scheduling experiments results using the actual processing times to perform the scheduling decisions. All the obtained nonlinear functions resulted in lower average slowdowns for all traces, with varying levels of improvements depending on the HPC



(a) Tasks characteristics generated from the Lublin and Feitelson [15] model, with processing time estimates generated from the Tsafrir *et al.* [23] model and $n_{max} = 256$ and with aggressive backfilling



(b) Tasks characteristics generated from the Lublin and Feitelson [15] model, with processing time estimates generated from the Tsafrir *et al.* [23] model and $n_{max} = 1024$ and with aggressive backfilling

Figure 6: Scheduling performance results with tasks characteristics generated from a workload model, with the addition of the aggressive backfilling, and using user estimated processing times in the scheduling decisions.

Table 5: Real workload traces used for evaluation of the scheduling policies.

Name	Year	# CPUs	# Jobs	Util %	Duration
Curie	2011	93,312	312,826	62.0	20 Months
ANL Interpid	2009	163,840	68,936	59.6	8 Months
SDSC Blue	2003	1,152	243,306	76.7	32 Months
CTC SP2	1997	338	77,222	85.2	11 Months

workload characteristics. More importantly, the difference between the upper and lower quartiles (box limits) was much lower when using the obtained nonlinear functions, meaning that the average slowdown was much more predictable and stable, a very desirable property for HPC systems. For all policies there were some cases where the average bounded slowdown was notoriously high, which occurred due to uncommon tasks characteristics in the traces.

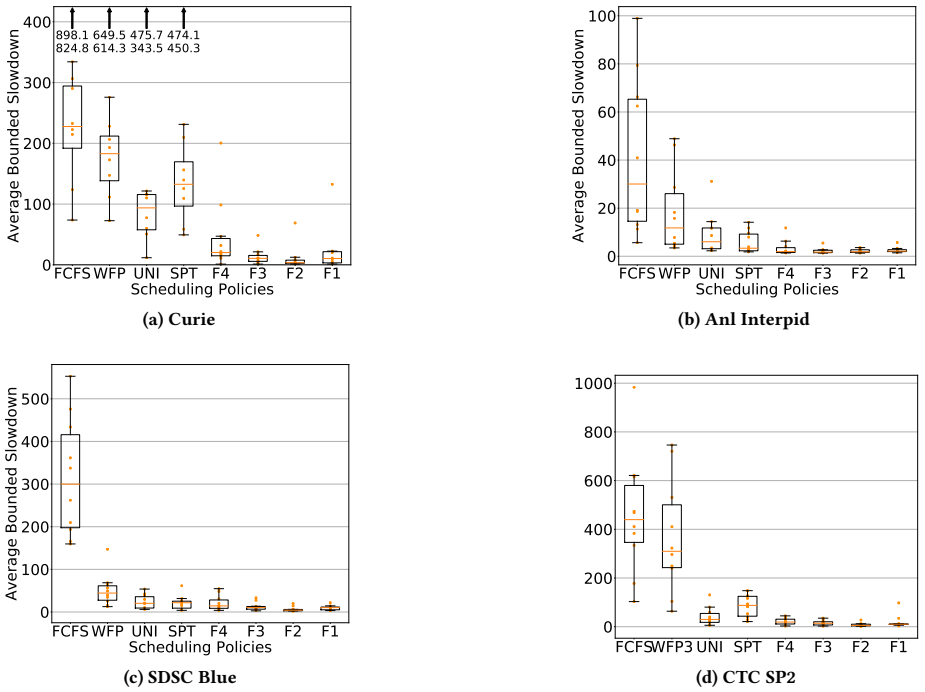
We can also observe that, with real workload traces, F1 was not the overall best policy. For example, the nonlinear function F2 achieved overall best results in the Curie, SDSC Blue and CTC SP2 workload traces, while the nonlinear function F3 achieved an overall best result for the Anl Interpid trace. This behavior is not unexpected, since the workload traces evaluated are highly different from each other and from the workload generation model used in the simulation scheme. But choosing any of the obtained policies F1 to F4 resulted in improvements in the median average slowdowns in all scenarios and smaller differences between the upper and lower quartiles in most scenarios.

4.3.2 Scheduling using user estimated task runtimes e. We evaluated the scheduling policies using the processing time estimate e obtained from the respective workload log when performing the scheduling decisions. Since the user estimates of the processing times are often rough and inaccurate, we expect a degradation in the performance of all scheduling policies. Figure 8 shows that the obtained functions F1 to F4 continued to generate lower median average slowdowns and differences between the upper and lower quartiles for all evaluated HPC platforms. Although the best function from F1 to F4 varied depending on the platform, any of them would result in significant performance improvements over existing *ad-hoc* policies. Moreover, the *ad-hoc* policies had either very large range of points or outlier points with average slowdowns much higher than the median, which can compromise the perceived QoS from the user point of view.

The results from this section are very impressive, considering the nonlinear functions F1 to F4 were trained using data from a single workload model in a simulated machine with 256 cores. These functions worked well as scheduling policies for HPC machines with highly different architectures, with up to 163,840 cores, very different workload types, and using inaccurate estimated task runtimes from real machine users.

4.3.3 Scheduling using estimated processing times and aggressive backfilling. In this experiment we considered the most realist scenario, where scheduling decisions are based on the user estimates of processing times e and with the addition of aggressive backfilling to reduce resource idleness. Figure 9 shows the corresponding dynamic scheduling experiments. Once again, the FCFS policy with backfilling (the EASY scheduling algorithm) was the scheduling policy which benefited the most with the introduction of backfilling. The performance results obtained for the WFP and UNI policies also reinforces the results obtained by Tang *et al.* [22], since we obtained similar comparative results for these scheduling policies, with the exception that we evaluated them with different workload logs.

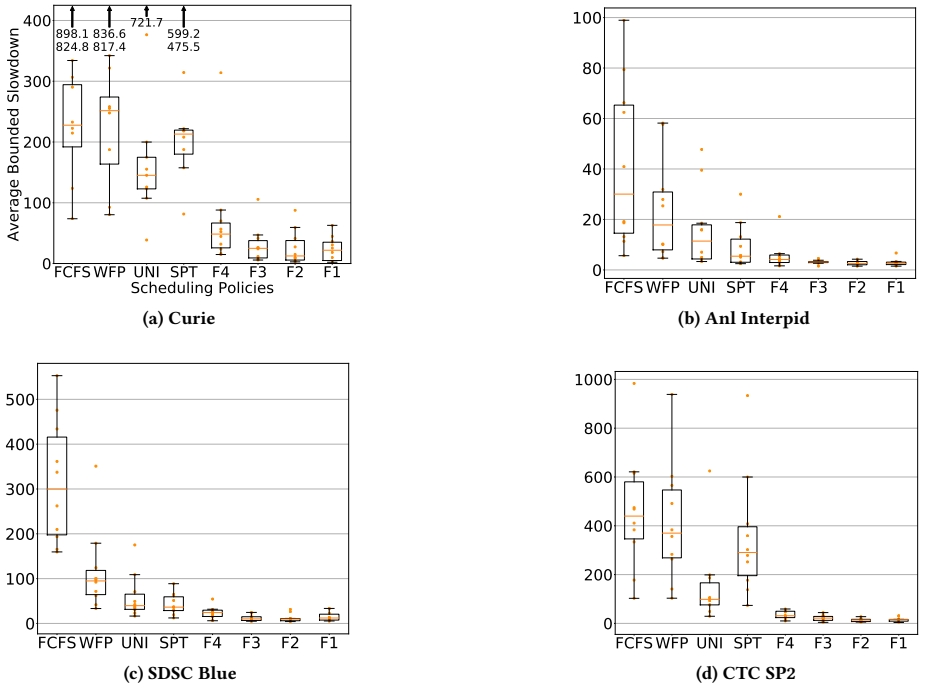
The obtained nonlinear functions had small benefits from using backfilling. This occurs because as tasks were already efficiently scheduled, there were less opportunities for backfilling tasks. Nevertheless, functions F1 to F4 still resulted in lower median average slowdowns and/or lower differences between the extreme quartiles for most scenarios, and continued to be a better general choice than the *ah-hoc* scheduling policies. It is still worth noting how well our general scheduling policies performed on these real traces of real HPC platforms using backfilling and user estimates of processing times.



(a) Curie (b) Anl Interpid

(c) SDSC Blue (d) CTC SP2

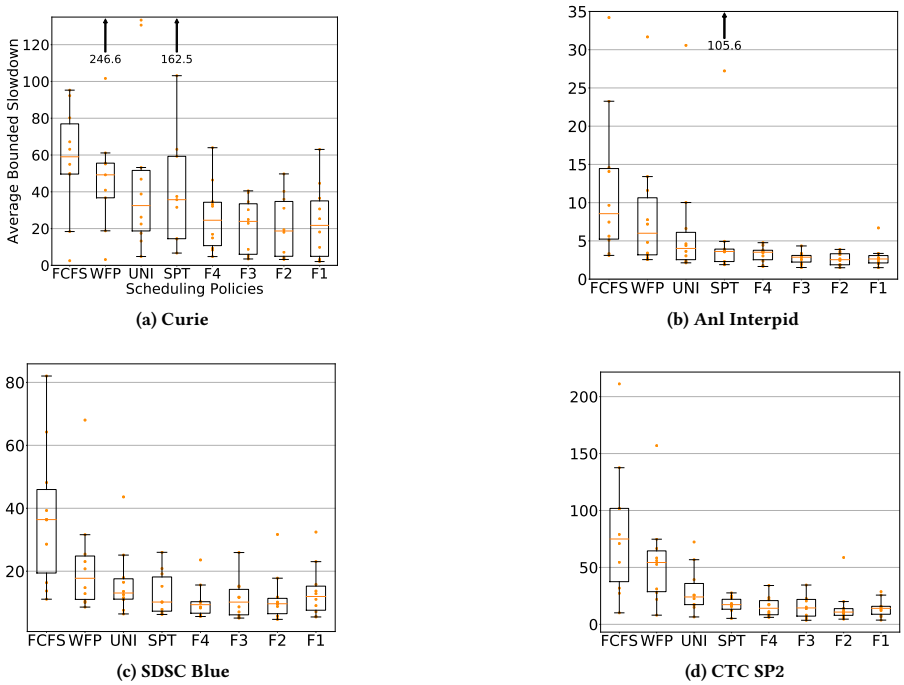
Figure 7: Scheduling performance results with tasks characteristics obtained from real HPC platform workload logs and using actual processing time in the scheduling decisions.



(a) Curie (b) Anl Interpid

(c) SDSC Blue (d) CTC SP2

Figure 8: Scheduling performance results with tasks characteristics obtained from real HPC platform workload logs and using user estimated processing times obtained from the same logs in the scheduling decisions.



(a) Curie (b) Anl Interpid

(c) SDSC Blue (d) CTC SP2

Figure 9: Scheduling performance results with tasks characteristics obtained from real HPC platforms, with the addition of the aggressive backfilling, and using user estimated processing times obtained from the same logs in the scheduling decisions.

5 CONCLUSIONS

Due to its simplicity, scheduling policies in the form of functions that take the tasks characteristics into consideration, plus the computationally inexpensive aggressive backfilling algorithm, are an appealing alternative for the problem of on-line scheduling of tasks in HPC platforms. In an equally simple manner, in this work we show that – by introducing a simulation procedure that captures the observations of the tasks scheduling behavior under several distinct conditions and a simple machine learning strategy to model these observations into nonlinear functions – we can obtain scheduling policies that effectively capture the scheduling behavior of the tasks and perform notoriously well, when compared to other classical and smart *ad-hoc* scheduling policies.

Using tasks characteristics obtained from the workload model of Lublin and Feitelson [15] and considering similar scenarios from the one used to capture the scheduling observations, the obtained scheduling policies achieved very impressive results. In the most realistic of these scenarios (*i.e.* using the aggressive backfilling in addition to the scheduling policies and using the processing time estimates to perform the scheduling decisions), the obtained scheduling policies achieved medians of the average bounded slowdowns at least 12.5 times better when compared to the best performing *ad-hoc* scheduling policy.

Although we used a workload model to generate the nonlinear functions, we could envision the same procedure being applied to obtain custom scheduling policies for a specific HPC platform, using its specific workload traces and architecture configurations. Our results using the workload model indicate that these custom policies could, in principle, generate important improvements in the achieved median of the average bounded slowdowns in these platforms.

However, using a workload model to capture the observations brought some important advantages. The generalized task properties present in the Lublin and Feitelson [15] workload model, when used to observe the scheduling behavior of the tasks in several distinct configurations, resulted in scheduling policies that are able to express efficient general patterns regarding to which task should selected for execution first. In this light, the obtained scheduling policies based on tasks from a workload model presented some generalization capabilities, showing consistent lower median values for the average bounded slowdown and being an overall better choice of scheduling policy in simulation experiments considering completely different task types and HPC platform configurations. Therefore, although it would be possible to define scheduling policies specifically for each HPC platform, as aforementioned stated, it seems that general policies, as the ones we found in this work, would be sufficient to efficiently schedule tasks for more specific workload types and HPC platform configurations.

As future work, we could improve the current work in two directions. The first would be to evaluate the scheduling policies by deploying them on real HPC platforms and check how they perform under real conditions. We used a simplified simulation to evaluate the obtained scheduling policies using real traces ignoring, for instance, network and memory bottlenecks that could appear from interactions among task execution. Nevertheless, we believe the simulations are a good approximation for these platforms.

The second direction is covering a wider range of HPC platforms. We plan to improve the strategy proposed by this work to obtain scheduling policies that also address the on-line scheduling of tasks in HPC platforms containing processing units with distinct architectures such as GPUs [18] and MICs [7], where multiple implementations, aiming a specific architecture, are available for the same task and the scheduler needs to select one of these implementations to be executed.

REFERENCES

[1] Anurag Agarwal, Selcuk Colak, Varghese S Jacob, and Hasan Pirkul. 2006. Heuristics and augmented neural networks for task scheduling with non-identical machines. *European Journal of Operational Research* 175, 1 (2006), 296–317.

[2] Derya Eren Akyol and G Mirac Bayhan. 2007. A review on evolution of production scheduling with neural networks. *Computers & Industrial Engineering* 53, 1 (2007), 95–122.

[3] Hadil Al-Daoud, Issam Al-Azzoni, and Douglas G Down. 2012. Power-aware linear programming based scheduling for heterogeneous computer clusters. *Future Generation Computer Systems* 28, 5 (2012), 745–754.

[4] Raymond J Carroll and David Ruppert. 1988. *Transformation and weighting in regression*. Vol. 30. CRC Press.

[5] Henri Casanova, Arnaud Giersch, Arnaud Legrand, Martin Quinson, and Frédéric Suter. 2014. Versatile, Scalable, and Accurate Simulation of Distributed Applications and Platforms. *J. Parallel and Distrib. Comput.* 74, 10 (June 2014), 2899–2917. http://hal.inria.fr/hal-01017319

[6] Adaptative Computing. 2017. Moab Workload Manager Documentation. http://www.adaptivecomputing.com/support/documentation-index/. (2017). Accessed: 2017-03-26.

[7] Alejandro Duran and Michael Klemm. 2012. The Intel® many integrated core architecture. In *High Performance Computing and Simulation (HPCS), 2012 International Conference on*. IEEE, 365–366.

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

[9] Dror G Feitelson, Larry Rudolph, Uwe Schwiegelshohn, Kenneth C Sevcik, and Parkson Wong. 1997. Theory and practice in parallel job scheduling. In *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer, 1–34.

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

[11] Christodoulos A Floudas and Xiaoxia Lin. 2005. Mixed integer linear programming in process scheduling: Modeling, algorithms, and applications. *Annals of Operations Research* 139, 1 (2005), 131–162.

[12] Y Georgiou. 2010. *Resource and Job Management in High Performance Computing*. Ph.D. Dissertation. PhD Thesis, Joseph Fourier University, France.

[13] Edwin SH Hou, Nirwan Ansari, and Hong Ren. 1994. A genetic algorithm for multiprocessor scheduling. *IEEE Transactions on parallel and distributed systems* 5, 2 (1994), 113–120.

[14] Eric Jones, Travis Oliphant, Pearu Peterson, and others. 2001–. SciPy: Open source scientific tools for Python. (2001–). http://www.scipy.org/

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

[16] Ahuva W. Mu'alem and Dror G. Feitelson. 2001. 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.

[17] Bill Nitzberg, Jennifer M Schopf, and James Patton Jones. 2004. PBS Pro: Grid computing and scheduling attributes. In *Grid resource management*. Springer, 183–190.

[18] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips. 2008. GPU computing. *Proc. IEEE* 96, 5 (2008), 879–899.

[19] Michael L. Pinedo. 2008. *Scheduling: Theory, Algorithms, and Systems* (3rd ed.). Springer Publishing Company, Incorporated.

[20] Harmel Karam Singh and Abdou Youssef. 1995. *Mapping and scheduling heterogeneous task graphs using genetic algorithms*. Master's thesis. George Washington University.

[21] Garrick Staples. 2006. TORQUE resource manager. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*. ACM, 8.

[22] Wei Tang, Zhiling Lan, Narayan Desai, and Daniel Buettner. 2009. Fault-aware, utility-based job scheduling on BlueGene/P systems. In *Cluster Computing and Workshops, 2009. CLUSTER'09. IEEE International Conference on*. IEEE, 1–10.

[23] Dan Tsafrir, Yoav Etsion, and Dror G Feitelson. 2005. Modeling user runtime estimates. In *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer, 1–35.

[24] Fatos Xhafa and Ajith Abraham. 2010. Computational models and heuristic methods for Grid scheduling problems. *Future generation computer systems* 26, 4 (2010), 608–621.

[25] Andy B Yoo, Morris A Jette, and Mark Grondona. 2003. Slurm: Simple linux utility for resource management. In *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer, 44–60.

A ARTIFACT DESCRIPTION: OBTAINING DYNAMIC SCHEDULING POLICIES WITH SIMULATION AND MACHINE LEARNING

A.1 Abstract

We provide the source code implementation of all of the strategies proposed in the paper as well as the source code of the simulation experiments used to evaluate our approach so the reader can (i) generate their own distribution $score(r, n, s)$, (ii) reproduce the nonlinear regression to obtain the scheduling policies presented in the paper or generate their own scheduling policies with their own generated distribution, and (iii) reproduce the dynamic scheduling experiments results presented in the paper.

A.2 Description

A.2.1 Check-list (artifact meta information).

- **Program: (i) Prototypes to generate the distribution $score(r, n, s)$, (ii) prototypes for the nonlinear function enumeration and nonlinear regression and (iii) the dynamic scheduling experiments prototypes.**
- **Compilation: GCC**
- **Data set: Workload models and real workload traces obtained from the Parallel Workload Archive:** http://www.cs.huji.ac.il/labs/parallel/workload/. **See more details below**.
- **Run-time environment: Python 2.7**
- **Hardware: Various x86 or x64 CPUs.**
- **Output: (i) Generated distribution $score(r, n, s)$, (ii) enumerated functions, their coefficients c_1, c_2 and c_3, and their respective fitness score, and (iii) dynamic scheduling experiments results with the four best obtained nonlinear functions presented in the paper.**
- **Experiment workflow: See below.**
- **Experiment customization: See below.**
- **Publicly available?: Yes.**

A.2.2 How software can be obtained. All source material can be downloaded at GitHub http://github.com/hpcsched/gen-sched-policies. The Git repository is structured in three prototypes represented by the following directories:

- `training-data-generator`: Contains all the source material to generate the distribution $score(r, n, s)$;
- `nonlinear-regression`: Contains all the source material to perform the nonlinear function enumeration and nonlinear regression;
- `sched-performance-tester`: Contains all source material to reproduce the dynamic scheduling experiments results present in the paper.

A.2.3 Hardware dependencies. Any modern x86 or x64 CPU is appropriate to execute the prototypes. It's advised, however, that the system has at least 6GB of RAM since some experiments consume high amounts of memory.

A.2.4 Software dependencies. The main software requirements are a Linux distribution (preferably Ubuntu or CentOS), the GCC C compiler and Python 2.7. Below is presented a list with the specific software requirements:

- **Simgrid 3.13 C libraries**: Publicly available at http://gforge.inria.fr/frs/download.php/file/35817/SimGrid-3.13.tar.gz. Installation instructions are available at http://simgrid.gforge.inria.fr/simgrid/3.13/doc/install.html;
- **ScyPy Python libraries**: Download and installation instructions available at https://www.scipy.org/install.html;
- **Matplotlib Python libraries**: Download and installation instructions available at http://matplotlib.org/users/installing.html. Required to reproduce the plots of the paper.

A.2.5 Datasets.

- Lublin and Feitelson workload model publicly available in: http://www.cs.huji.ac.il/labs/parallel/workload/models.html#lublin99
- Tsafrir *et al.* processing time estimate model publicly avilable at: http://www.cs.huji.ac.il/labs/parallel/workload/models.html#tsafrir05
- CEA Curie workload log publicly avilable at: http://www.cs.huji.ac.il/labs/parallel/workload/l_cea_curie/index.html
- ANL Iterpid workload log publicly avilable at: http://www.cs.huji.ac.il/labs/parallel/workload/l_anl_int/index.html
- SDSC Blue workload log publicly avilable at: http://www.cs.huji.ac.il/labs/parallel/workload/l_sdsc_blue/index.html
- CTC SP2 workload log publicly avilable at: http://www.cs.huji.ac.il/labs/parallel/workload/l_ctc_sp2/index.html

A.3 Installation

Most of the source programs of this work are coded in Python and therefore no installation or compilation is necessary. However, the scheduling simulation programs are coded in C and they need to be compiled. Therefore, after the prototypes are downloaded from the repository (see Section A.2.2) and the software dependencies are installed (see Section A.2.4), it is necessary to perform the following commands from the root repository of the prototypes:

```
$ cd training-data-generator
$ make
$ cd ../sched-performance-tester
$ make
```

A.4 Experiment workflow

A.4.1 Workflow 1: Generating the distribution $score(r, n, s)$. To generate your own distribution $score(r, n, s)$ in the same way that it was performed in the paper, once the source codes is properly compiled (see Section A.3), perform the following commands from the root repository of the prototypes:

```
$ cd training-data-generator
$ nohup python generate_simulation_data.py &
```

The `nohup` command starts a background Python process that continuously generates tuples of tasks (S, Q) and simulate the trials of these tuples to obtain the distribution $score(r, n, s)$. It is recommended to leave this process running at least for a couple of days so enough scheduling observations are performed in the simulations. From the `training-data-generator` directory there are two important directories: The `task-sets` directory contains all the task tuples (S, Q) generated. Each file present

in this directory is a CSV file in which each line contain characteristics (`runtime,#processors,submit_time`) of a task. In its turn, the `training-data` directory contains all of the trial score distributions generated. Each file in this directory is a CSV file in which each line represents the observed scheduling behavior (`runtime,#processors,submit_time,score`) of a task. To join all of the trial score distributions to make the distribution $score(r, n, s)$, perform the following command in the same directory of the previous command:

```
$ python gather_data.py
```

This command creates a file called `score-distribution.csv` which contains the distribution $score(r, n, s)$ obtained from the simulations.

A.4.2 Workflow 2: Enumerating nonlinear functions and performing the fit with nonlinear regression. To reproduce the nonlinear regression and obtain the nonlinear functions presented in the paper, from the root directory of the prototypes perfom the following commands:

```
$ cd nonlinear-regression
$ python nlr_scipy_enumerate_functions.py \
        score-distribution.csv
```

The output of this command is the enumerated functions, their coefficients c_1, c_2 and c_3 and their respective fitness value, in decreasing order of fitness. More details about the output can be found in Section A.5. The `score-distribution.csv` file is the file that we used to obtain the nonlinear functions presented in the paper. This file can be changed with another distribuition $score(r, n, s)$ generated (see Section A.4.1).

A.4.3 Workflow 3: Reproducing the dynamic scheduling experiments results. To reproduce the dynamic scheduling experiments results presented in the paper, perform the following commands from the root repository of the prototypes:

```
$ cd sched-performance-tester
$ python test_all.py
```

The `test_all.py` is a wrapper script that calls all of the Python scrips present in the `sched-performance-tester` directory, performing all the dynamic scheduling experiments presented in the paper and outputs the statistics of the experiments (medians, means and standard deviations) plus the resulting plot similar to the ones presented in the paper. Alternatively, it's possible to execute each Python script present in the `sched-performance-tester` directory individually to perform only a specific experiment.

A.5 Evaluation and expected result

A.5.1 Generating the distribution $score(r, n, s)$ output. The output of this part is a CSV file named `score-distribution.csv` containing the observations of the scheduling behavior captured in the simulations. Each line of this file contains the characteristics and the resulting scheduling behavior (`runtime,#processors,submit_time,score`) of a task. Below is an example of the expected output:

```
50.0,8.0,88224.0,0.0347251055192
3.0,4.0,88302.0,0.0292281817457
7298.0,58.0,88334.0,0.0350921606481
98.0,1.0,88350.0,0.0333329252836
27.0,9.0,88356.0,0.0307780390597
11.0,32.0,88414.0,0.0278089667369
8758.0,8.0,88421.0,0.031821251468
```

A.5.2 Enumerating nonlinear functions and performing the fit with nonlinear regression. The output of this part is the enumerated functions, their coefficients c_1, c_2 and c_3 and their respective fitness value, in decreasing order of fitness. One example of a expected output is presented below:

```
(-0.0155183403 x log10(runtime)) *
(-0.0005149209 x id(#cores)) +
(0.0069596182 x log10(submit)),
fitness=0.0052776
```

Note that algebraic equivalent functions can be enumerated and, in this case, their fitness value will be equal.

A.5.3 Reproducing the dynamic scheduling experiments results. The output of this part is the result statistics of the dynamic scheduling experiments performed. For one experiment, an expected output is the following:

```
Performing scheduling performance test for
the workload trace lublin_256.
Configuration:
Using actual runtimes, backfilling disabled
Experiment Statistics:
Medians:
FCFS=5846.87 WFP=3630.67 UNI=1799.74 SPT=943.59
F4=583.89 F3=89.94 F2=29.66 F1=29.58

Means:
FCFS=6194.92 WFP=3716.46 UNI=2336.92 SPT=1415.13
F4=721.77 F3=593.65 F2=143.40 F1=134.93

Standard Deviations:
FCFS=3321.45 WFP=2908.26 UNI=2050.01 SPT=1539.56
F4=700.10 F3=853.46 F2=227.07 F1=264.51
Boxplot saved in file plots/model_256_r.pdf
```

As mentioned in the output, A boxplot similar with the ones presented in the paper with the respective results is generated and stored in the `plots` subdirectory.