A new on-line method for scheduling independent tasks

Giorgio Lucarelli, Fernando Machado Mendonca, Denis Trystram

A new on-line method for scheduling independent tasks

Giorgio Lucarelli, Fernando Machado Mendonca, and Denis Trystram
Univ. Grenoble Alpes, CNRS, Inria, LIG, F-38000 Grenoble France
{*giorgio.lucarelli,fernando.machado-mendonca,denis.trystram*}*@imag.fr*

***Abstract*—We present a new method for scheduling independent tasks on a parallel machine composed of identical processors. This problem has been studied extensively for a long time with many variants. We are interested here in designing a generic algorithm in the on-line non-preemptive setting whose performance is good for various objectives. The basic idea of this algorithm is to detect some problematic tasks that are responsible for the delay of other shorter tasks. Then the former tasks are redirected to be executed in a dedicated part of the machine. We show through an extensive experimental campaign that this method is effective and in most cases is closer to some standard lower bounds than the base-line method for the problem.**

***Keywords*-on-line scheduling, non-preemptive, independent tasks, stretch, redirections**

I. Introduction

We are interested in studying the problem of scheduling a set of independent sequential *without preemption*. These tasks are submitted on a multi-core parallel machine. This is a basic scheduling problem whose different variants have been studied for different objectives. The aim is to determine good solutions in short (polynomial) time, whose objective values are close to (a lower bound of) the optimal solution.

Most existing works, and particularly theoretical studies, consider *off-line* executions, that correspond to scenarios where the entire instance is known in advance. In this case, the algorithms naturally target the minimization of objectives like the maximum completion time (*makespan*) or the total completion time of all tasks. Several provably good algorithms have been proposed for these objectives which achieve constant approximation ratios.

In this paper, we consider more realistic scenarios where the tasks and their characteristics are only known when they are submitted in the system. This *on-line* problem is harder since we should be able to make decisions with a partial knowledge of the instance. In this context, objectives like makespan have no concrete meaning since they could strongly depend on the maximum submission time. For these reason, in the on-line setting we usually consider objectives based on the *flow-time* which corresponds to the time that a task remains to the system. We are mainly interested in the *stretch* of the tasks (also known as *slow-down*) which normalizes the flow-time of each task with respect to its processing time.

Our purpose within this work is to introduce a new technique for on-line scheduling independent tasks which will be applied to optimize several enhanced objectives. The main idea of this technique is to detect a reasonable number of tasks that have a negative impact on the whole execution and to *redirect* them to a *dedicated pool* of processors in order to be executed apart from the other tasks. More precisely, we propose to split the set of tasks into *common* and *heavy* tasks. This split is performed on-line at run time. Informally, a task will be called heavy if its execution delays an important number of shorter tasks. Whenever a task is characterized as heavy, it is redirected to the pool of dedicated processors and this decision will never be reconsidered in the future.

The dedicated pool consists of a subset of the available processors of the machine, that is no additional processors are used. However, it will be only used by the heavy tasks. The size of this pool should be related to several parameters, and specifically the size of the machine and parameters that cause the redirection of the heavy tasks.

In order to characterize a task as heavy, we propose to keep track of the number of shorter tasks that arrive during the execution of a task. For this, a counter is initialized at the beginning of the execution of a task, and if this counter reaches a given threshold, then the task is characterized as heavy and it is redirected. The execution of heavy tasks on the dedicated processors is restarted from the beginning, so that the non-preemptive assumption to be satisfied. The above idea of the counter has been used in [17], where a theoretical upper bound has been proved. However, this algorithm completely rejects the heavy tasks instead of redirecting them, an action that is not permissible in real systems.

Based on the observation that most heavy tasks in the above policy have quite long processing times, we also propose a random method which characterizes a long task as heavy with some given probability. This method avoids the interruption of the tasks and the re-execution of part of them, but does not benefit of the information about the currently delayed tasks provided by the counter.

Both above ideas have been assessed by an extensive simulation campaign based on real data coming from actual execution traces. The experimental results show the significant benefits compared to standard base-line policies for scheduling tasks without preemptions.

In Section II we formally define the studied problem and we give the notations used throughout the paper, while

in Section III we describe known results and position this paper in regards to related works. In Section IV we first present the base-line non-preemptive algorithms as well as the preemptive lower bounds that will be used in our experiments. Then, we propose our algorithms based on heavy tasks and redirections. In Section V we describe the construction of experimental data which are extracted from 7 different traces. Next, we provide the results of the experimental campaign consisting of the parameter tuning of our new methods as well as their comparison in the constructed instances with the base-line algorithms and the lower bounds. Finally, we conclude in Section VI.

II. Definitions and notations.

In the following, we consider a set of n independent sequential tasks and a parallel machine consisting of m processors. Each task should be executed on a single processor without *preemptions*. We denote by M_i, $1 \leq i \leq m$, each processor of this machine. Each task T_j, $1 \leq j \leq n$, is characterized by a *processing time* p_j, which becomes known to the scheduler only after its submission at time r_j; we call r_j the *release time* of T_j. In what follows we denote by $p_j(t)$ the remaining processing time of the task T_j at time t. Note that $p_j(r_j) = p_j$.

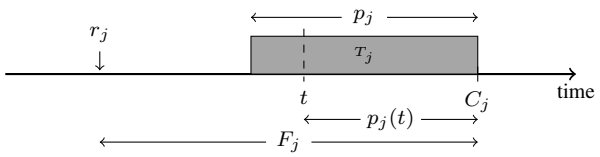


Figure 1: Summary of notations.

Given a schedule, C_j denotes the *completion time* of T_j, that is the time at which it is completed. From a more realistic perspective and taking into account the user point of view, a natural measure of the quality of service delivered to a task is the amount of time it spends in a system. The basic objective in this direction is the *flow-time* (also called response time) of a task, which is defined as the amount of time it remains in the system until being completed. In other words, the flow-time of a task is composed by the time this task waits in the system to begin its execution plus its processing time. The flow-time of a task T_j is denoted by $F_j = C_j - r_j$.

Many variants of flow-time metric have been investigated in different settings. Since the flow-time depends on the processing time, it varies a lot for different tasks. For this reason the *stretch* (or slowdown) metric has been proposed which normalizes the flow-time of a task by dividing it by its processing time, *i.e.* $S_j = {}^{F_j}/_{p_j}$. This metric is often used in fair scheduling where the users are willing to wait longer for long tasks as opposed to short ones. However, the stretch metric has also some drawbacks since it largely focus on very short tasks. Specifically, such a short task cannot wait even for a natural period of time because its stretch will be exploded. To overcome this, the stretch metric has been refined to the *bounded stretch* (or bounded slowdown) metric [13] which is defined as $\frac{F_j}{\max\{p_j, B\}}$, where $B > 1$ is a small constant (we set $B = 10$). In this paper we are mainly interested in the bounded stretch metric, and secondarily in flow-time. For simplicity, we will henceforth refer to bounded stretch by just stretch.

For both stretch and flow-time, we consider three different objective functions. The first objective function concerns the minimization of the maximum stretch or flow-time over all tasks. These objectives are denoted by $S_{\max}$ and $F_{\max}$, respectively. The second objective function corresponds to the minimization of the total stretch or flow-time for all tasks, *i.e.* $\sum S_j$ and $\sum F_j$, respectively. A third objective function that balances the previous two functions, while it is also considered to increase the fairness among tasks, is the norm. We will consider the second norms of stretch and flow-time which are defined as $(\sum S_j^2)^{1/2}$ and $(\sum F_j^2)^{1/2}$, respectively. Note that the first norm corresponds to the sum function, while the infinite norm to the max function.

III. Related works

Scheduling efficiently concurrent tasks on a parallel machine is a central problem for reaching good performances. There exist a huge literature on this problem focusing on standard objectives based on completion times (see for example [11]). Most of these studies are done in an off-line setting. Here, we are interested in flow-time based objectives that are more appropriate for the non-preemptive and mainly for the on-line settings. In general, the flow-oriented objectives are harder than the standard makespan or total completion time objectives.

In what follows, we say that an off-line algorithm achieves a ρ-approximation ratio if the objective value of its constructed solution is at most ρ times bigger than the objective value of an optimal solution. In a similar way, we may define the competitive ratio for on-line algorithms, comparing the algorithm's solution with the off-line optimal solution.

If preemptions are allowed, the classical Shortest Remaining Processing Time policy (SRPT) provides an optimal solution for minimizing the total flow-time and a 2-competitive solution for minimizing the total stretch on a single processor [8]. Moreover, a variant of SRPT achieves a competitive ratio of 13 for the total stretch minimization problem on parallel processors [18]. Furthermore, Bender *et al.* [9] proposed a variant of the classical Earliest Deadline First (EDF) policy for the max stretch objective on a single processor which leads to a $\mathcal{O}(\sqrt{\Delta})$-competitive algorithm, where Δ is the ratio of the maximum over the minimum processing time of the instance. As we consider non-preemptive scheduling, the above results may only serve as lower bounds.

In the non-preemptive case, Kellerer *et al.* [14] showed that there exists a lower bound of $\Omega(n^{\frac{1}{2}-\epsilon})$ on the approximability of total flow-time minimization problem even on a single processor, while the corresponding lower bound for the on-line case is $\Omega(n)$ [10]. On the positive side, an $\mathcal{O}(\sqrt{\frac{n}{m}}\log\frac{n}{m})$-approximation algorithm has been presented in [16] for the problem of minimizing the total flow-time on m parallel processors. Moreover, Weighted Shortest Processing Time (WSPT) is a $\mathcal{O}(\Delta+\frac{3}{2}-\frac{1}{2m})$-competitive algorithm [21] for the more general problem where each task has a weight and the objective is to minimize the total weighted flow-time. Since this problem is a generalization of the total stretch minimization problem, to which reduces if the weight of a task is equal to the inverse of its processing time, the result holds also for the latter objective.

Concerning the max function, it is known that First-Come-First-Served (FCFS) is optimal on a single processor and $(3-\frac{2}{m})$-competitive on parallel processors for the max flow-time minimization problem [9]. Bender *et al.* [9] showed that the max stretch minimization problem on parallel processors cannot be approximated within a factor of $\Omega(\Delta^{\frac{1}{3}})$ on a single processor and $\Omega(n^{1-\epsilon})$ on parallel processors, unless $\mathcal{P}=\mathcal{NP}$. Legrand *et al.* [15] studied the FCFS policy for the problem of minimizing the max stretch on a single processor and they proved that it achieves a $\mathcal{O}(\Delta)$-competitive ratio. This result has been improved to $(\frac{\sqrt{5}-1}{2}\Delta+1)$ in [12]. No results are known for the max stretch objective in multi-processors setting.

Let us conclude this section by a remark: the idea presented in this paper is related to the over-provisioning mechanism, which is, for instance, discussed in [19] in the context of energy saving in high-performance parallel platforms. The idea is to add extra hardware (processors) in order to improve the power utilization as well as the task throughput in power-constrained platforms. In our case, we propose to dedicate a part of the processors to a specific utilization on a small number of problematic tasks for improving the whole execution of the bunch of tasks.

IV. Algorithms

In this section we propose new policies for scheduling sequential tasks on parallel processors based on the idea of redirecting the "hard" tasks to a dedicated subset of processors. Before this, we briefly describe several standard scheduling policies whose efficiency is compared in Section V with the efficiency of our algorithms.

In all cases, we will consider two basic paradigms: the *common queue* and the *immediate dispatch* models. In the first model, we use a single common queue Q for all processors in order to store the *pending* tasks, i.e., the tasks that are released but not yet executed. Then, whenever a processor becomes idle we select a task from Q according to rules that depend on the specific policy. In the second model, we use a different queue Q_i for each processor M_i and we dispatch each task just upon its arrival to one of these queues. We will describe the *dispatching policy* in Section IV-D, after presenting the scheduling policies.

A. Standard scheduling policies

Initially, we describe the policies based on a common queue Q for all processors. Specifically, we will consider the following three "global" policies.

First-Come First-Served (FCFS): Whenever a processor becomes idle, schedule on it the earliest released task in Q.

Global Shortest Processing Time (GSPT): Whenever a processor becomes idle, schedule on it the shortest processing time task in Q.

Global Shortest Remaining Processing Time (GSRPT): Whenever a processor becomes idle, schedule on it the shortest remaining processing time task in Q. At the arrival of a new task T_j we search for the processor that actually executes the task with the longest remaining processing time. Let M_i be this processor and T_k be the task executed on M_i at r_j. If $p_j \leq p_k(r_j)$ then we interrupt the execution of T_k and we start executing T_j on processor M_i, while the task T_k is added again to Q with processing time $p_k(r_j)$.

Next, we describe the immediate dispatch versions of the SPT and SRPT policies, respectively. Recall that the dispatching policy will be presented in Section IV-D.

Shortest Processing Time (SPT): Whenever a processor M_i becomes idle, schedule on it the shortest processing time task in Q_i.

Shortest Remaining Processing Time (SRPT): Whenever a processor M_i becomes idle, schedule on it the shortest remaining processing time task in Q_i. At the arrival of a new task T_j which is dispatched to processor M_i, let T_k be the task which is actually executed by M_i. If $p_j \leq p_k(r_j)$ then we interrupt the execution of T_k and we start executing T_j on processor M_i, while the task T_k is added again to Q_i with processing time $p_k(r_j)$.

Note that the policies based on SRPT produce preemptive schedules, and they are only used as lower bounds. Moreover, all the above policies apart from FCFS prioritize the tasks with the shortest (remaining) processing time. The motivation to this is due to the fact that this order provides good schedules for both the total flow-time and the total stretch objectives. Recall that, for the single processor case, SRPT is a preemptive policy which is optimal for total flow-time [8] and almost optimal for total stretch, while SPT is an optimal non-preemptive policy for both objectives if there are no release dates [20]. On the other hand, it is well-known that FCFS is an optimal non-preemptive policy for the maximum flow-time objective on a single processor.

B. Scheduling with redirections

In this section we propose new algorithms for scheduling non-preemptively a set of sequential tasks on a multi-processor machine. The basic idea is to detect the *heavy*

tasks whose non-preemptive execution increases significantly the waiting time of shorter tasks. These heavy tasks will be then *redirected* to be executed on a small subset of processors reserved exclusively for them.

In order to give the intuition of our idea, let us present an example that shows why any on-line non-preemptive policy $\mathcal{A}$ can be arbitrarily bad with respect to an optimal non-preemptive schedule. For simplicity, we consider that a single processor is available. Assume that a long task T_0 of processing time p is released and, without loss of generality, the scheduler starts executing T_0 at time 0. Then, p unit processing time tasks are released: task T_i, $1 \leq i \leq p$, is released at time $r_i = i$. Since $\mathcal{A}$ is a non-preemptive algorithm, it cannot interrupt the long task and hence executes the short tasks during the interval $[p+1, 2p+1]$ as, for example, shown in Figure 2. The total flow-time of any such schedule is $O(p^2)$. On the other hand, the offline optimal non-preemptive schedule will delay the execution of the long task and execute all short tasks before T_0, which gives a schedule of total flow-time only $O(p)$.

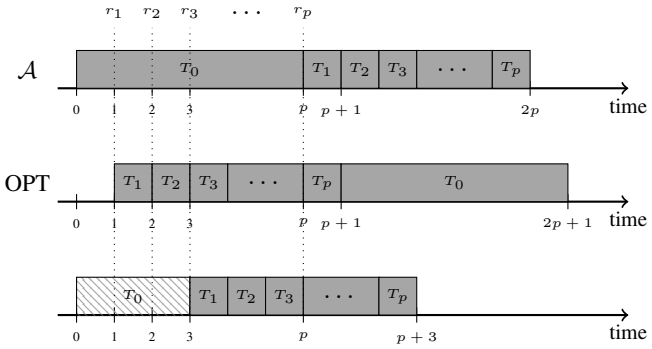


Figure 2: Intuition of redirection.

Motivated by this example, we propose to characterize a task T_j as *heavy* if the number of shorter tasks that arrive during its execution attains a given threshold τ. When this threshold is attained, the execution of T_j is interrupted and it is redirected to a small subset of processors dedicated only for heavy tasks. Note that, the execution of any heavy task on these processors will be restarted from the beginning, that is heavy tasks are not preemptively executed but they are interrupted and restarted. The last part of Figure 2 demonstrates an example of the resulting schedule if the threshold is set to be $\tau = 3$. In this case, the task T_0 is interrupted at time $r_3 = 3$ while the remaining tasks can start their execution and complete earlier than in the schedule of $\mathcal{A}$ but later than in the optimal schedule. In this way, the value of the threshold could describe a trade-off between the number of the tasks that should be restarted and the performance for the remaining tasks.

As in the case of the standard scheduling policies, we will consider two versions of our method, one with a common queue and one with immediate dispatch. The formal definition of these two versions follows. In both cases, we consider that a threshold τ is given.

Global Shortest Processing Time with Redirections (GSPT-RD): Whenever a processor becomes idle, schedule on it the shortest processing time task in Q. At the beginning of the execution of a task T_k on any processor M_i we introduce a counter c_k which is initialized to zero. At the arrival of a new task T_j, we search for the processor that actually executes the task with the longest remaining processing time. Let M_i be this processor and T_k be the task executed on M_i at r_j. If $p_j < p_k(r_j)$ then we increase by one the counter c_k. If $c_k = \tau$ then we interrupt T_k and we redirect it to the set of dedicated processors.

Shortest Processing Time with Redirections (SPT-RD): Whenever a processor M_i becomes idle, schedule on it the shortest processing time task in Q_i. At the beginning of the execution of a task T_k on any processor M_i we introduce a counter c_k which is initialized to zero. At the arrival of a new task T_j which is dispatched to processor M_i during the execution of T_k, we increase by one the counter c_k only if $p_j < p_k(r_j)$. If $c_k = \tau$ then we interrupt T_k and we redirect it to the set of dedicated processors.

It remains to describe the scheduling policy that we use for the subset of the processors dedicated for the redirected tasks. In fact, in each case we use the same policy as for the main set of processors, without however allowing new redirections for the already redirected tasks. In other words, the GSPT (resp., SPT) policy is used for the dedicated processors along with the GSPT-RD (resp., SPT-RD) policy.

Remark that the above methods based on redirections are not comparable with the preemptive methods SRPT and GSRPT, even though all of them may interrupt the execution of a task. First, SRPT and GSRPT may interrupt a task several times, in contrast with SPT-RD and GSPT-RD which interrupt each task once and redirect it to the dedicated processors where interruption is not allowed. Second, SRPT and GSRPT continue the execution of a preempted task from the point of its interruption, while SPT-RD and GSPT-RD restart the interrupted jobs. For these reasons, SRPT and GSRPT can be used only as lower bounds.

C. Scheduling with random redirections

In some applications, the interruption of a task is very complicated even if it will be re-executed from the beginning. For example, this is the case of output-intensive applications. For this kind of applications, we propose to substitute the threshold rule used in the previous section with a random rule which will be applied before the beginning of the execution of each task. In this way, no interruptions will be performed.

In order to simulate the threshold rule, we observe that the tasks redirected based on it are in general long tasks. For this reason, we characterize a task as *potentially heavy*

if its processing time is at least $b \cdot p_{\max}$, where $b \in [0,1]$ is a constant that characterizes how long is the task and $p_{\max}$ is the maximum processing time over all tasks that have been arrived by the current time. Then, a potentially heavy task will be redirected with a probability $c \in [0,1]$. Note that a significant drawback of the random rule with respect to the threshold rule is that the first one only cares about the load of the task under consideration, while the second one relates the load of this task with the current load of the system.

We again consider two versions of the random method which are formally defined as follows.

Global Shortest Processing Time with Random Redirections (GSPT-RR): Whenever a processor becomes idle, schedule on it the shortest processing time task in Q. At the arrival of a new task T_j, if $p_j \geq b \cdot p_{\max}$ then select a number c_j uniformly at random in $[0,1]$. If $c_j \leq c$ then redirect T_j to the set of dedicated processors. In any other case, add T_j in Q.

Shortest Processing Time with Random Redirections (SPT-RR): Whenever a processor M_i becomes idle, schedule on it the shortest processing time task in Q_i. At the arrival of a new task T_j, if $p_j \geq b \cdot p_{\max}$ then select a number c_j uniformly at random in $[0,1]$. If $c_j \leq c$ then redirect T_j to the set of dedicated processors. In any other case, apply the dispatching policy for the task T_j.

D. Dispatching policy

In order to decide to which processor we will dispatch each new task, we use a rule that tries to balance the flow-times of tasks over all processors. Specifically, for each processor separately, we compute the *marginal increase* in the total flow-time that causes the potential assignment of the new task to this processor. Then, we dispatch the new task to the processor for which this increase is minimum. Note that this decision is irrevocable, except for the redirection case, but still the processors used for redirected tasks are apart.

In what follows in this section, we clearly define the marginal increase in different cases. Observe first that in any of the presented scheduling policies which are based on the immediate dispatch model (SPT, SRPT and SPT-RD), the tasks are executed in shortest (remaining) processing time order. Based on this observation, we can say that the marginal increase that causes a new task T_j if it is dispatched to a processor M_i consists of the flow-time of T_j as well as the increase of the flow-time for the tasks of processing time longer than p_j in the queue Q_i. Assuming that the task T_k is executed at r_j on M_i, the marginal increase can be defined in the general case as follows:

$$\Delta\mathcal{F}_i = \left(p_k(r_j) + \sum_{\substack{T_\ell \in Q_i: \\ p_\ell(r_j) \leq p_j}} p_\ell(r_j) + p_j\right) + \sum_{\substack{T_\ell \in Q_i: \\ p_\ell(r_j) > p_j}} p_j$$

where the first part is the potential flow-time of T_j if it is dispatched to M_i, while the second part corresponds to the potential delay for the tasks of longer (remaining) processing time than T_j. Note that, in the cases of SPT and SPT-RD, for the value of $p_\ell(r_j)$ in the first sum, it always holds that $p_\ell(r_j) = p_\ell$.

The above general definition has two exceptions. First, in the case of SRPT and only if the potential dispatching of the new task T_j on M_i will preempt the executed task T_k, i.e., if $p_j < p_k(r_j)$, then the marginal increase is defined as follows:

$$\Delta\mathcal{F}_i = p_j + \sum_{\substack{T_\ell \in Q_i \cup \{T_k\}: \\ p_\ell(r_j) > p_j}} p_j$$

Second, in the case of SPT-RD and only if the potential dispatching of the new task T_j on M_i will redirect the executed task T_k, i.e., if $p_j < p_k(r_j)$ and $c_k = \tau - 1$, then the marginal increase is defined as follows:

$$\Delta\mathcal{F}_i = \left(\sum_{\substack{T_\ell \in Q_i: \\ p_\ell(r_j) \leq p_j}} p_\ell(r_j) + p_j\right) + \sum_{\substack{T_\ell \in Q_i: \\ p_\ell(r_j) > p_j}} p_j - \sum_{T_\ell \in Q_i} p_k(r_j)$$

where the negative term corresponds to the decrease in the flow-time of all tasks in Q_i by the removal of the remaining processing time of T_k.

V. Experimental results

In this section we describe the generation of the instances that we will use to experimentally compare our new methods based on redirections with the state-of-the-art algorithms for scheduling independent tasks.

A. Construction of instances

In our experiments we use data from 7 traces in order to validate the results across different scenarios. The traces used and their characteristics are shown in the first part of Table I. Since these traces consist of parallel independent jobs, we next describe how to meaningfully transform them to construct instances of sequential independent tasks.

For each trace we create 20 instances. Specifically, in order to create an instance we uniformly at random select an initial point t during the whole time horizon of the corresponding trace. Then, all jobs that are released in the interval $[t, t+s]$ will belong to the instance, where s is the instance size (total duration) as this is given for the different traces in the last column of Table I. Let n be the number of all selected jobs for an instance and m be the total number of processors used by the platform in the corresponding original trace.

Each job is characterized by a *real execution time* e_j, a *number of required processors* q_j and a *release date* r_j. We consider only the real execution times since user submitted execution times are strongly overestimated. As an example, one third of the jobs in the *Curie* trace have user submitted execution times of 24 hours which corresponds to the default setting of the system. In order to transform a parallel job to

trace	traces characteristics			instances characteristics (in average)		
	# processors	# tasks	size in days	# processors	# tasks	size in days
curie [1]	80640	279991	6089	160	70215	60
hpc2n [2]	240	201998	1268	17	4516	30
kth-sp2 [7]	100	20483	333	12	860	15
llnl-thunder [3]	4096	97875	148	48	4601	7
metacentrum [4]	806	86586	157	18	4155	7
ricc [5]	8192	431547	153	200	19401	7
sdsc-blue-4.1 [6]	1152	195587	979	27	6263	30

Table I: Traces and instances characteristics. The processors and tasks numbers are the average over the 20 instances created for each trace.

a sequential task, we used the following three methods. Note that, the main characteristic of all of them is that they keep the ratio of the total execution volume over the number of processors for the created instance the same as in the initial trace.

A. For each parallel job, create q_j sequential tasks each one of processing time e_j.
B. For each parallel job, create a sequential task with processing time $p_j = q_j \cdot e_j$.
C. Let $Q = \left(\sum_{j=1}^{n} q_j\right)/n$ be the average number of requested processors for all jobs of the instance. For each parallel job, create a sequential task with processing time $p_j = \frac{q_j}{Q} \cdot e_j$. Moreover, the number of processors that will be used for this instance is m/Q.

We have performed some initial experiments for these three methods. However, methods A and B have significant drawbacks. Method A generates many tasks, and it can create instances that are too big to be kept in memory. Moreover, it constructs instances where multiple tasks have the same processing times. Method B overcomes both previous drawbacks. Although it creates a single sequential task per parallel job with the same execution volume, it disturbs the relations between the execution times of jobs. For example, in an instance created by method B, one job of short execution time and big number of required processors and another job of long execution time and small number of required processors could lead to tasks of the same processing times, or even worse change their ordering with respect to their execution times. For this reason, in the experimental campaign that we present in the following section, the instances are created using method C. The only (small) disadvantage of this method is that affects also the number of processors in the created instances, in order to keep the ratio of the total load over the number of processors unchanged. In general, the number of processors in each instance can be significantly smaller than the number of processors used by the platform in the corresponding original trace. However, this could be considered as a natural assumption when dealing with sequential tasks.

B. Experiments

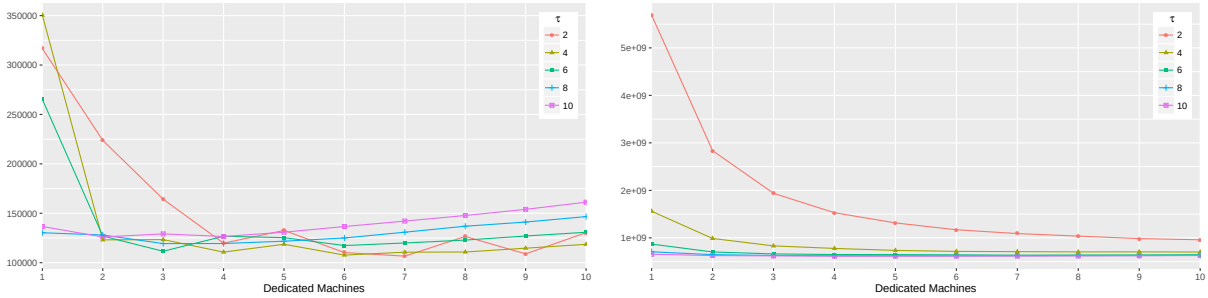
All the algorithms have been implemented in a custom discrete event simulator. The code of the simulator and the experimental data used can be found at http://fernando.mendonca.xyz/redirection.tar.gz. All values given in the following subsections and figures correspond to the average over the 20 instances constructed as described above for each trace.

Tuning of parameters: The first part of our experiment concerns the tuning of the parameters used for methods SPT-RD, GSPT-RD, SPT-RR and GSPT-RR. Specifically, for the methods SPT-RD and GSPT-RD we have to define the number of dedicated processors m_d used for the heavy tasks and the threshold τ which determines the characterization of a task as heavy. Recall that the dedicated processors is a subset of the processors of the machine and hence $m_d < m$. In other words, the basic set of processors used for the common tasks contains $m - m_d$ processors.

For *Curie* and *RICC* traces we will set $\tau \in \{1, 2, \ldots, 10\}$, while for the remaining instances we set $\tau \in \{1, 2, \ldots, 5\}$. Moreover, we use $m_d \in \{1, 2, \ldots, 10\}$ and test all combinations of parameters.

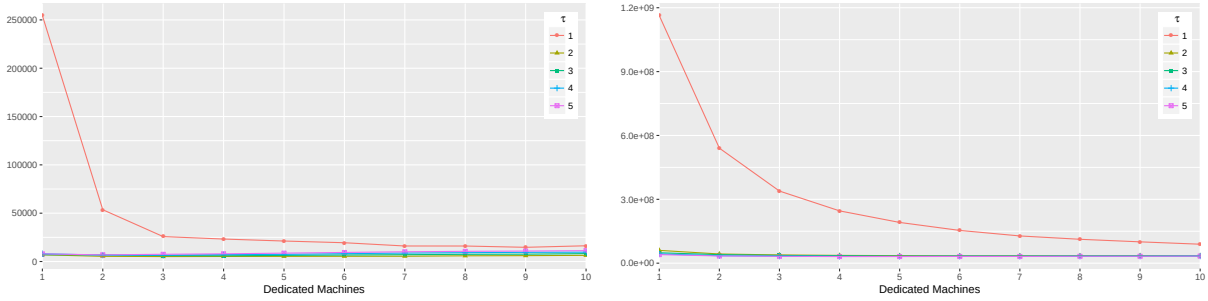
In Figure 3, the values of total stretch and total flow-time of SPT-RD for different parameters are shown. We present here only the two more representative traces: *Curie* and *LLNL-Thunder*. In the *Curie* trace, we observe that while the stretch is, in general, increasing with the number of dedicated processors for $m_d \geq 4$, the flow-time decreases. This is not true for the *LLNL-Thunder* trace, but a similar situation is observed for this trace with the threshold increase (see the forth couple of graphs in Figure 3). In general, we have observed a very variant relation between the two parameters and the different objectives. The value of the threshold directly affects the number of redirected tasks, while indirectly may affect the under-/over-utilization of the dedicated or the basic set of processors. For this reason, the two parameters are strongly related and they should be selected in conjunction.

Similar observations can be done for SPT-RR and GSPT-RR methods. Here we have three parameters: the number of dedicated processors m_d, the parameter b that determines how long is a task and the probability of redirection c. In order to deal with the parameter b, we first observed that all traces have a very small number of very long tasks. To see this, we calculated the ratio of the $0.8n$ longest task of each



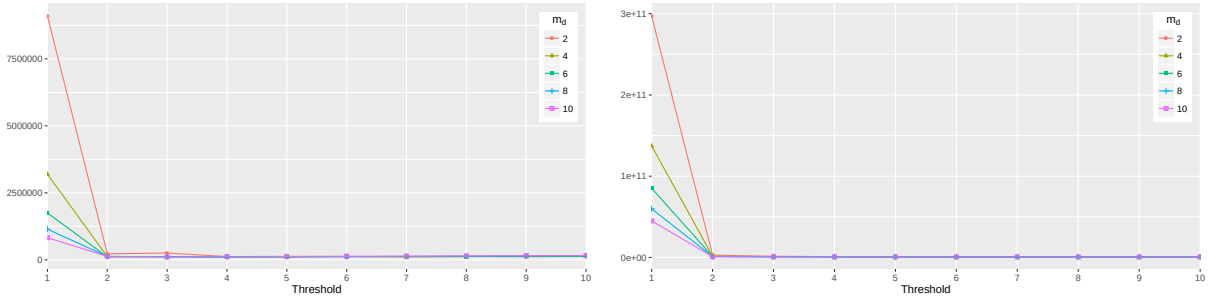
(a) Curie – total stretch

(b) Curie – total flow-time



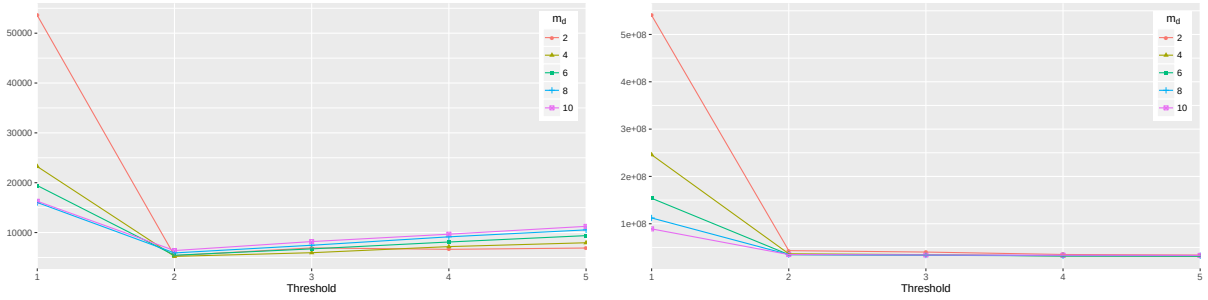
(c) LLNL-Thunder – total stretch

(d) LLNL-Thunder – total flow-time



(e) Curie – total stretch

(f) Curie – total flow-time



(g) LLNL-Thunder – total stretch

(h) LLNL-Thunder – total flow-time

Figure 3: Total stretch (on the left) and total flow-time (on the right) for *Curie* and *LLNL-Thunder* traces. In the four first graphs, the horizontal axis corresponds to the number of dedicated processors m_d while each line corresponds to a threshold τ. The inverse situation appears in the four latter graphs.

trace over the longest task of it, and this is equal to 0.02 in average for all traces. For this reason, we fix $b = 0.02$. For the other two parameters we did a similar analysis using the values $m_d \in \{1, 2, \ldots, 10\}$ and $c = \{0.2, 0.4, 0.6, 0.8, 1\}$.

In Table II, we present the "best" combination of parameters for each trace and method which will be also used in the next section. These parameters are chosen in such a way that each method has a good (or average) performance for all objectives. Another choice could be the parameters that minimize a particular objective. However, this choice can have the opposite effect for the other objectives as it is indicated in Figure 3.

Trace	SPT-RD		GSPT-RD		SPT-RR		GSPT-RR	
	m_d	τ	m_d	τ	m_d	c	m_d	c
Curie	10	10	10	4	9	0.8	10	0.8
HPC2N	5	2	10	1	5	1	10	1
KTH-SP2	8	1	7	1	2	0.8	3	0.8
LLNL-Thunder	7	2	10	5	8	1	9	1
Metacentrum	4	5	3	4	3	0.4	2	0.2
RICC	10	5	10	10	5	0.2	5	0.2
SDSC-Blue-4.1	5	2	10	1	10	1	8	0.8

Table II: Selected parameters for each trace

Comparison with base-line algorithms and lower bounds: The second part of our experiment is the comparison between the implemented methods SPT, GSPT, SRPT, GSRPT, SPT-RD, GSPT-RD, SPT-RR and GSPT-RR. We do not include here the results for the standard FCFS method since its performance in most cases is significantly worse, and perturbs the readability of the figures.

We start by focusing on the *Curie* trace which is the largest trace of our collection. The comparison of the algorithms for the different metrics can be found in Figures 4a to 4f. These Figures show that although SPT is stable for the flow metric, it is highly detrimental to the stretch metric with regard to the lower bounds (SRPT and GSRPT). On the other hand, the redirection of tasks improve the stretch metrics for the *Curie* trace. Specifically, there is a significant decrease in all stretch metrics for the GSPT-RD, SPT-RR and GSPT-RR methods. The reason that the same improvement cannot be seen for the SPT-RD method is the presence of bursts of short tasks in the *Curie* trace, which causes the redirection even of tasks that can be characterized as short to the dedicated processors and the consequent over-utilization of these processors. In this case, the best solution is to use a bigger number of dedicated processors which would be able to execute redirected tasks more efficiently. The tendency shown in Figure 3(d) can also justify that the flow metric would see significant improvement if more dedicated processors are used.

Next, we consider the remaining traces using only the stretch metric as can be seen in Figures 5 to 9. These Figures show that in most of the cases there is a significant improvement in all stretch metrics if redirection is used comparing to the SPT and GSPT methods. In addition, we can see that in some cases the stretch metric for the methods that employ redirections is even close to the SRPT and GSRPT methods, justifying the efficiency of our methods.

VI. Conclusion

We have presented a method for on-line scheduling independent tasks on a multi-processor machine based on the characterization of a small number of tasks as heavy, that is problematic for the whole performance of the system. These tasks are executed on a dedicated subset of processors. We proposed a deterministic and a random approach for detecting the heavy tasks. These methods are evaluated on instances extracted by seven real traces. The experimental results show the dependency of both methods on the correct choice of the parameters. Both methods focus mainly on stretch metrics at the expense of flow-time metrics. In general, the deterministic method outperforms the standard SPT policy and in many cases its performance approached the performance of the preemptive policy SRPT which can be considered as lower bound. The gains for the random method are smaller but, in several cases, significant comparing again with SPT.

The most intriguing future direction is to extend the above ideas for parallel jobs. However, there are several issues that should be considered. First, the size of the dedicated processors part should be large enough for dealing with wide jobs, which could lead to under-utilization of these processors. A solution to this could be to leave the borders between dedicated and normal processors more free. Second, the counter method is not anymore clear how it will be applied since a new job may need to cause the redirection of more than one running jobs. Third, the commonly used backfilling strategy should also be able to benefit from redirections. In other words, redirection could be used to increase the available space (holes) in order to backfill more efficiently.

Acknowledgement

This paper was partly funded by the CAPES Foundation, Ministry of Education, Brazil.

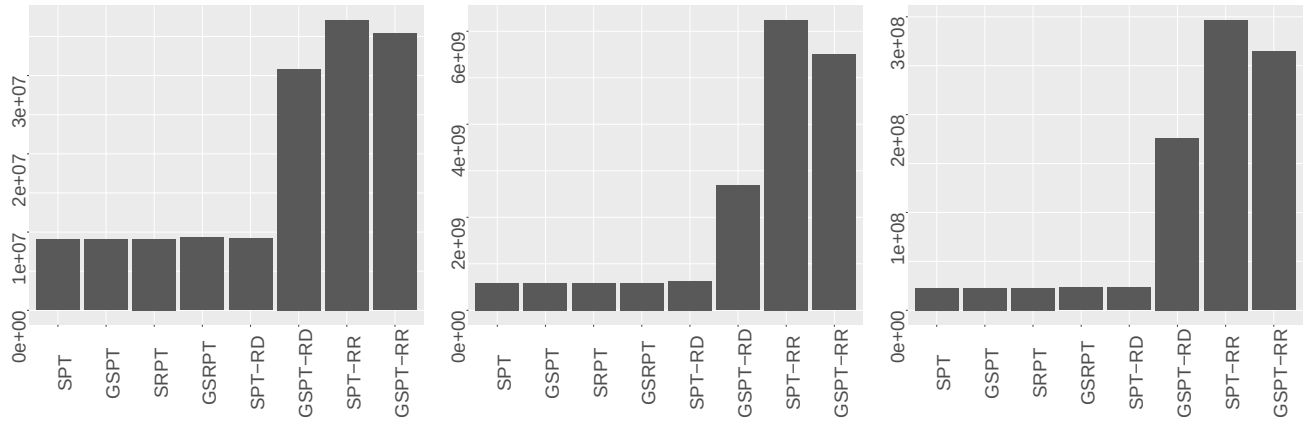
Additionally, experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see https://www.grid5000.fr).

References

[1] The cea curie log. http://www.cs.huji.ac.il/labs/parallel/workload/l_cea_curie/index.html. Accessed: 2016-11-29.

[2] The hpc2n seth log. http://www.cs.huji.ac.il/labs/parallel/workload/l_hpc2n/index.html. Accessed: 2016-11-29.

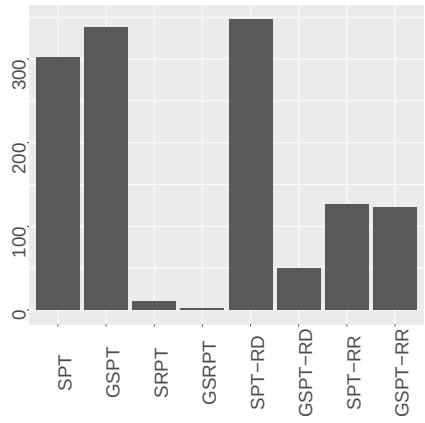
[3] The llnl thunder log. http://www.cs.huji.ac.il/labs/parallel/workload/l_llnl_thunder/index.html. Accessed: 2016-11-29.



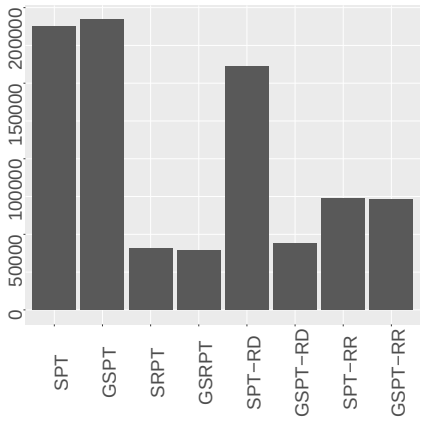
(a) Max flow

(b) Total flow

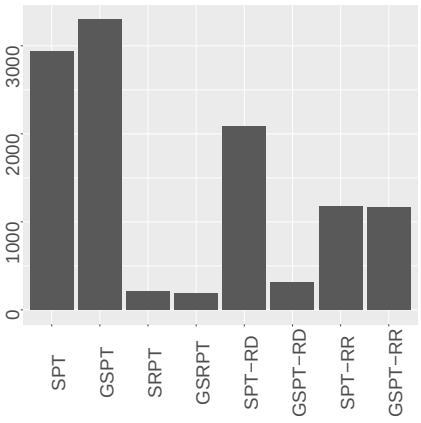
(c) Flow norm



(d) Max stretch

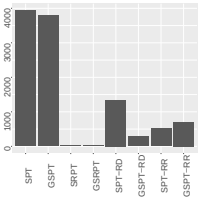


(e) Total stretch

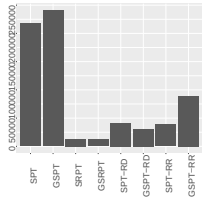


(f) Stretch norm

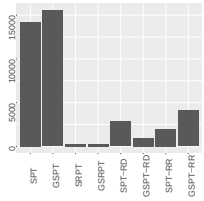
Figure 4: Curie



(a) Max stretch

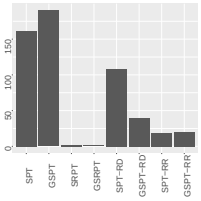


(b) Total stretch

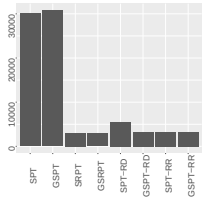


(c) Stretch norm

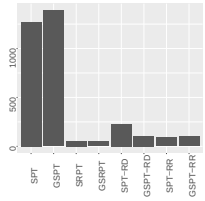
Figure 5: HPC2N



(a) Max stretch

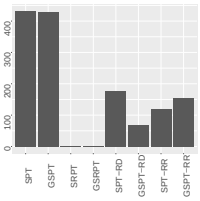


(b) Total stretch

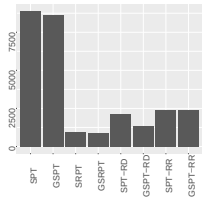


(c) Stretch norm

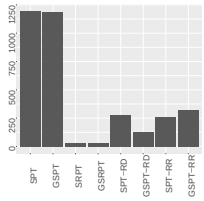
Figure 7: LLNL-Thunder



(a) Max stretch

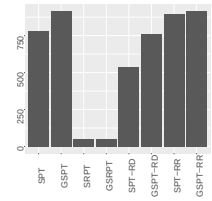


(b) Total stretch

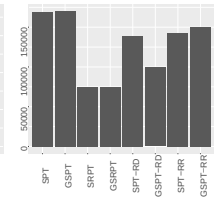


(c) Stretch norm

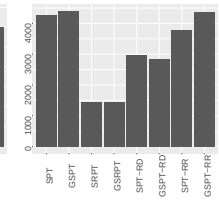
Figure 6: KTH-SP2



(a) Max stretch

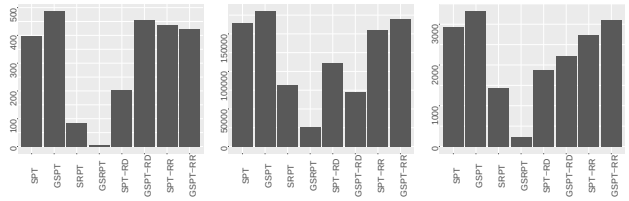


(b) Total stretch



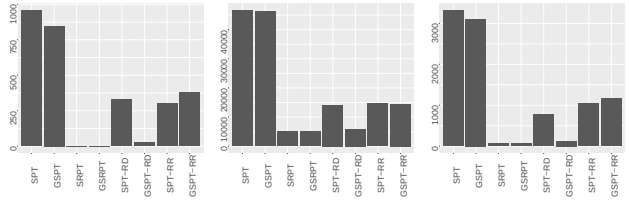
(c) Stretch norm

Figure 8: Metacentrum



(a) Max stretch (b) Total stretch (c) Stretch norm

Figure 9: RICC



(a) Max stretch (b) Total stretch (c) Stretch norm

Figure 10: SDSC-Blue-4.1

[4] The metacentrum log. http://www.cs.huji.ac.il/labs/parallel/workload/l_metacentrum/index.html. Accessed: 2016-11-29.

[5] The ricc log. http://www.cs.huji.ac.il/labs/parallel/workload/l_ricc/index.html. Accessed: 2016-11-29.

[6] The san diego supercomputer center (sdsc) blue horizon log. http://www.cs.huji.ac.il/labs/parallel/workload/l_sdsc_blue/index.html. Accessed: 2016-11-29.

[7] The swedish royal institute of technology (kth) ibm sp2 log. http://www.cs.huji.ac.il/labs/parallel/workload/l_kth_sp2/index.html. Accessed: 2016-11-29.

[8] K. R. Baker. *Introduction to Sequencing and Scheduling*. Wiley, New York, 1974.

[9] Michael A. Bender, Soumen Chakrabarti, and S. Muthukrishnan. Flow and stretch metrics for scheduling continuous job streams. In *ACM-SIAM Symposium on Discrete Algorithms*, pages 270–279, 1998.

[10] Chandra Chekuri, Sanjeev Khanna, and An Zhu. Algorithms for minimizing weighted flow time. In *ACM Symposium on Theory of Computing (STOC)*, pages 84–93, 2001.

[11] Maciej Drozdowski. *Scheduling for Parallel Processing*. Computer Communications and Networks. Springer, 2009.

[12] Pierre-François Dutot, Erik Saule, Abhinav Srivastav, and Denis Trystram. Online non-preemptive scheduling to optimize max stretch on a single machine. In *International Conference on Computing and Combinatorics (COCOON)*, volume 9797 of *LNCS*, pages 483–495. Springer, 2016.

[13] Dror G. Feitelson. Metrics for parallel job scheduling and their convergence. In *Job Scheduling Strategies for Parallel Processing (JSSPP)*, volume 2221 of *LNCS*, pages 188–206. Springer, 2001.

[14] Hans Kellerer, Thomas Tautenhahn, and Gerhard J. Woeginger. Approximability and nonapproximability results for minimizing total flow time on a single machine. *SIAM J. Comput.*, 28(4):1155–1166, 1999.

[15] Arnaud Legrand, Alan Su, and Frédéric Vivien. Minimizing the stretch when scheduling flows of divisible requests. *J. Scheduling*, 11(5):381–404, 2008.

[16] S. Leonardi and D. Raz. Approximating total flow time on parallel machines. *Journal of Computer and System Sciences*, 73:875–891, 2007.

[17] Giorgio Lucarelli, Nguyen Kim Thang, Abhinav Srivastav, and Denis Trystram. Online non-preemptive scheduling in a resource augmentation model based on duality. In *European Symposium on Algorithms (ESA)*, volume 57 of *LIPIcs*, pages 63:1–63:17. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016.

[18] S. Muthukrishnan, Rajmohan Rajaraman, Anthony Shaheen, and Johannes Gehrke. Online scheduling to minimize average stretch. *SIAM J. Comput.*, 34(2):433–452, 2004.

[19] Osman Sarood, Akhil Langer, Abhishek Gupta, and Laxmikant V. Kalé. Maximizing throughput of overprovisioned hpc data centers under a strict power budget. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 807–818. IEEE, 2014.

[20] Wayne E. Smith. Various optimizers for single-stage production. *Naval Research Logistics Quarterly*, 3(1-2):59–66, 1956.

[21] J. Tao and T. Liu. WSPT's competitive performance for minimizing the total weighted flow time: From single to parallel machines. *Mathematical Problems in Engineering*, 10.1155/2013/343287, 2013.