Online Parallel Scheduling of Non-uniform Tasks: Trading Failures for Energy*

Antonio Fernández Anta[1], Chryssis Georgiou[2], Dariusz R. Kowalski[3]**, and Elli Zavou[1,4]***

[1] Institute IMDEA Networks
[2] University of Cyprus
[3] University of Liverpool
[4] Universidad Carlos III de Madrid

Abstract. Consider a system in which tasks of different execution times arrive continuously and have to be executed by a set of processors that are prone to crashes and restarts. In this paper we model and study the impact of parallelism and failures on the competitiveness of such an online system. In a fault-free environment, a simple Longest-in-System scheduling policy, enhanced by a redundancy-avoidance mechanism, guarantees optimality in a long-term execution. In the presence of failures though, scheduling becomes a much more challenging task. In particular, no parallel deterministic algorithm can be competitive against an offline optimal solution, even with one single processor and tasks of only two different execution times. We find that when additional energy is provided to the system in the form of processor speedup, the situation changes. Specifically, we identify thresholds on the speedup under which such competitiveness cannot be achieved by any deterministic algorithm, and above which competitive algorithms exist. Finally, we propose algorithms that achieve small bounded competitive ratios when the speedup is over the threshold.

Keywords: Scheduling, Non-uniform Tasks, Failures, Competitiveness, Online Algorithms, Energy Efficiency.

1 Introduction

Motivation. In recent years we have witnessed a dramatic increase on the demand of processing computationally-intensive jobs. Uniprocessors are no longer capable of coping with the high computational demands of such jobs. As a result, multicore-based parallel machines such as the K-computer [32] and Internet-based supercomputing platforms such as SETI@home [23] and EGEE Grid [16] have become prominent computing environments. However, computing in such environments raises several challenges. For example, computational jobs (or tasks) are injected dynamically and continuously, each job may have different computational demands (e.g., CPU usage or processing time) and the processing elements are subject to unpredictable failures. Preserving power consumption is another challenge of rising importance. Therefore, there is a corresponding need for developing algorithmic solutions that would efficiently cope with such challenges.

* This research was supported in part by the Comunidad de Madrid grant S2009TIC-1692, Spanish MICINN/MINECO grant TEC2011-29688-C02-01, and NSF of China grant 61020106002.
** This work was performed during the visit of D. Kowalski to Institute IMDEA Networks
*** Partially supported by FPU Grant from MECD

Table 1. Summary of results.

Condition	Number of task costs	Task competitiveness	Cost competitiveness	Algorithm
$s < c_{max}/c_{min}$ and $s < \frac{\gamma c_{min}+c_{max}}{c_{max}}$	≥ 2	∞	∞	Any
$s \geq c_{max}/c_{min}$	Any	1	c_{max}/c_{min}	(n, β)-LIS
$\frac{\gamma c_{min}+c_{max}}{c_{max}} \leq s < \frac{c_{max}}{c_{min}}$	2	1	1	γn-Burst
$s \geq 7/2$	Finite	c_{max}/c_{min}	1	LAF

Much research has been dedicated to task scheduling problems, each work addressing different challenges (e.g., [9, 13–15, 17, 18, 20, 22, 26, 31, 12]). For example, many works address the issue of dynamic task injections, but do not consider failures (e.g., [11, 21]). Other works consider scheduling on one machine (e.g., [3, 27, 30]); with the drawback that the power of parallelism is not exploited (provided that tasks are independent). Other works consider failures, but assume that tasks are known a priori and their number is bounded (e.g., [5, 7, 12, 18, 22]), where other works assume that tasks are uniform, that is, they have the same processing times (e.g., [12, 17]). Several works consider power-preserving issues, but do not consider, for example, failures (e.g., [10, 11, 31]).

Contributions. In this work we consider a computing system in which tasks of *different* execution times arrive *dynamically and continuously* and must be performed by a set of n processors that are prone to *crashes and restarts*. Due to the dynamicity involved, we view this task-performing problem as an online problem and pursue competitive analysis [28, 2]. Efficiency is measured as the maximum *pending cost* over any point of the execution, where the pending cost is the sum of the execution times of the tasks that have been injected in the system but are not performed yet. We also account for the maximum *number of pending tasks* over any point of the execution. The first measure is useful for evaluating the remaining processing time required from the system at any given point of the computation, while the second for evaluating the number of tasks still pending to be performed, regardless of the processing time needed.

We show that no parallel algorithm for the problem under study is competitive against the best off-line solution in the classical sense, however it becomes competitive if static processor *speed scaling* [6, 4, 11] is applied in the form of a *speedup* above a certain threshold. A speedup s means that a processor can perform a task s times faster than the task's system specified execution time (and therefore has a meaning only when $s \geq 1$). Speed scaling impacts the *energy consumption* of the processor. As a matter of fact, the power consumed (i.e., the energy consumed per unit of time) to run a processor at a speed x grows superlinearly with x, and it is typically assumed to have a form of $P = x^{\alpha}$, for $\alpha > 1$ [31, 1]. Hence, a speedup s implies an additional factor of $s^{\alpha-1}$ in the power (and hence energy) consumed. The use of a speedup is a form of resource augmentation [25].

Our investigation aims at developing competitive online algorithms that require the smallest possible speedup. As a result, one of the main challenges of our work is to identify the speedup thresholds, under which competitiveness cannot be achieved and

over which it is possible. In some sense, our work can be seen as investigating the trade-offs between *knowledge* and *energy* in the presence of failures: How much energy (in the form of speedup) does a deterministic online scheduling algorithm need in order to match the efficiency (i.e., to be competitive with) of the optimal offline algorithm that possesses complete knowledge of failures and task injections? (It is understood that there is nothing to investigate if the offline solution makes use of speed-scaling as well). Our contributions are summarized as follows (see Table 1):

Formalization of fault-tolerant distributed scheduling: In Section 2, we formalize an online task performing problem that abstracts important aspects of today's multicore-based parallel systems and Internet-based computing platforms: dynamic and continuous task injection, tasks with different processing times, processing elements subject to failures, and concerns on power-consumption. To the best of our knowledge, this is the first work to consider such a version of dynamic and parallel fault-tolerant task scheduling.

Study of offline solutions: In Section 3, we show that an offline version of a similar task-performing problem is NP-hard, for both pending cost and pending task efficiency, even if there is no parallelism (one processor) and the information of all tasks and processor availability is known.

Necessary conditions for competitiveness: In Section 4, we show *necessary* conditions (in the form of threshold values) on the value of the speedup s to achieve competitiveness. To do this, we need to introduce a parameter γ, which represents the smallest number of c_{min}-tasks that an algorithm can complete (using speedup s), in addition to a c_{max}-task, such that the offline algorithm cannot complete more tasks in the same time. Note that c_{min} and c_{max} are lower and upper bounds on the cost (execution time) of the tasks injected in the system.

We propose two conditions, (a) $s < \frac{c_{max}}{c_{min}}$, and (b) $s < \frac{\gamma c_{min}+c_{max}}{c_{max}}$ and show that if *both* hold, then *no* deterministic sequential or parallel algorithm is competitive when run with speedup s. It is worth noting that this holds even if we only have a single processor, and therefore this result could be generalized for stronger models that use centralized or parallel scheduling of multiple processors. Observe that, satisfying condition (b) implies $\rho > 0$, which automatically means that condition (a) is also satisfied.

Sufficient conditions for competitiveness: Then, we design two scheduling algorithms, each matching a different threshold bound from the necessary conditions above, showing *sufficient* conditions on s that lead to competitive solutions. In fact, it can be shown that in order to have competitiveness, it is sufficient to set $s = c_{max}/c_{min}$ if $c_{max}/c_{min} \in [1, \varphi]$, and $s = 1 + \sqrt{1 - c_{min}/c_{max}}$ if otherwise, where $\varphi = \frac{1+\sqrt{5}}{2}$ is the golden ratio.

Algorithm (n, β)-LIS: For the case when condition (a) does not hold (i.e., $s \geq \frac{c_{max}}{c_{min}}$), we develop algorithm (n, β)-LIS, presented in Section 5. We show that, under these circumstances, (n, β)-LIS is 1-pending-task-competitive and $\frac{c_{max}}{c_{min}}$-pending-cost-competitive for parameter $\beta \geq \frac{c_{max}}{c_{min}}$ and for any given number of processors n. These results hold for any collection of tasks with costs in the range $[c_{min}, c_{max}]$.

Algorithm γn-Burst: It is not difficult to observe that algorithm (n, β)-LIS cannot be competitive when condition (a) holds but condition (b) does not (i.e., $\frac{\gamma c_{min}+c_{max}}{c_{max}} \leq s < \frac{c_{max}}{c_{min}}$). For this case we develop algorithm γn-Burst, presented in Section 6. We

show that when tasks of two different costs, c_{min} and c_{max}, are injected, the algorithm is both 1-pending-task and 1-pending-cost competitive.

These results fully close the gap with respect to the conditions for competitiveness on the speedup in the case of two different task costs, establishing $s = \min\{\frac{c_{max}}{c_{min}}, \frac{\gamma c_{min}+c_{max}}{c_{max}}\}$ as the threshold for competitiveness.

Algorithm LAF, low energy guaranteed: In Section 7, we develop algorithm LAF that is again competitive for the case when condition (b) does not hold, but in contrast with γn-Burst, it is more "geared" towards pending cost efficiency and can handle tasks of multiple different costs. We show that this algorithm is competitive for speedup $s \geq \frac{7}{2}$. Hence, unlike the above mentioned algorithms, its competitiveness is with respect to a speedup that is independent of the values c_{max} and c_{min}.

Omitted proofs and further details can be found in the full version [8].

Task Scheduling. We assume the existence of an entity, called *Shared Repository* (whose detailed specification is given in Section 2), that abstracts the service by which clients submit computational tasks to our system and that notifies them when they are completed. This allows our results to be conceptually general, instead of considering specific implementation details. The Shared Repository is *not* a scheduler, since it does not make any task allocation decisions; processors simply access this entity to obtain the set of pending tasks. Such an entity, and implementations of it, have been considered, for example, in the Software Components Communication literature, where it is referred as the Shared Repository Pattern (see for example [24, 29], and references therein).

This makes our setting simpler, easier to implement and more scalable than other popular settings with stronger scheduling computing entities, such as a *central scheduler*. Note that even in the case of the central scheduler, a central repository would still be needed in order for the scheduler to keep track of the pending tasks and proceed with task allocation. Hence, the underline difference of our setting with that of a central scheduler is that in the latter, scheduling decisions and processing is done by a single entity which allocates the tasks to the processors, as opposed to our setting where scheduling decisions are done in *parallel* by the participating processors for deciding what task each processor should perform next. As a consequence, all the results of our work also hold for such stronger models: algorithms work not worse than in the Shared Repository setting since it is a weaker model. The necessary conditions on energy threshold also hold as they are proven for a scenario with a single processor, where these two models are indistinguishable.

Related Work. The work most closely related to this work is the one by Georgiou and Kowalski [17]. As in this work, they consider a task-performing problem where tasks are dynamically and continuously injected to the system, and processors are subject to crashes and restarts. Unlike this work, the computation is broken into synchronous rounds and the notion of *per-round* pending-task competitiveness is considered instead. Furthermore, tasks are assumed to have *unit cost*, i.e., they can be performed in one round. The authors consider at first a central scheduler and then show how and under what conditions it can be implemented in a message-passing distributed setting (called local scheduler). They show that even with a central scheduler, no algorithm can be competitive if tasks have different execution times. This result has essentially moti-

vated the present work; to use speed-scaling and study the conditions on speedup for which competitiveness is possible. As it turns out, extending the problem for tasks with different processing times and considering speed-scaling is a non-trivial task; different scheduling policies and techniques had to be devised.

Our work is also related with studies of parallel online scheduling using identical machines [26]. Among them, several papers consider speed-scaling and speedup issues. Some of them, unlike our work, consider dynamic scaling (e.g., [4, 10, 11]). Usually, in these works preemption is allowed: an execution of a task may be suspended and later restarted from the point of suspension. In our work, the task must be performed from scratch. The authors of [19] investigate scheduling on m identical speed-scaled processors without migration (tasks are not allowed to move among processors). Among others, they prove that any z-competitive online algorithm for a single processor yields a zB_a-competitive online algorithm for multiple processors, where B_a is the number of partitions of a set of size a. What is more, unlike our work, the number of processors is not bounded. The work in [6] considers tasks with deadlines (i.e., real-time computing is considered), but no migration, whereas the work in [4] considers both. We note that none of these works considers processor failures. Considering failures, as we do, makes parallel scheduling a significantly more challenging problem.

2 Model and Definitions

Computing Setting. We consider a system of n homogeneous, fault-prone processors, with unique ids from the set $[n] = \{1, 2, \ldots, n\}$. We assume that processors have access to a shared object, called *Shared Repository* or *Repository* for short. It represents the interface of the system that is used by the clients to submit computational tasks and receive the notifications about the performed ones.

Operations. The data type of the repository is a set of tasks (to be described later) that supports three operations: *inject, get,* and *inform*. The *inject* operation is executed by a client of the system, who adds a task to the current set, and as discussed below, this operation is controlled by an adversary. The other two operations are executed by the processors. By executing a *get* operation, a processor obtains from the repository the set of *pending tasks*, i.e., the tasks that have been injected into the system, but the repository has not been notified that they have been completed yet. To simplify the model we assume that, if there are no pending tasks when the *get* operation is executed, it blocks until some new task is injected, and then it immediately returns the set of new tasks. Upon computing a task, a processor executes an *inform* operation, which notifies the repository about the task completion. Then the repository removes this task from the set of pending tasks. Note that due to processor crashes, it would not be helpful for a processor to notify the repository of the task it has scheduled before actually performing the task. Each operation performed by a processor is associated with a point in time (with the exception of a *get* that blocks) and the outcome of the operation is instantaneous (i.e., at the same time point).

Processor cycles. Processors run in *real-time cycles*, controlled by an algorithm. Each cycle consists of a *get* operation, a computation of a task, and an *inform* operation (if a task is completed). Between two consecutive cycles an algorithm may choose to have a processor idling for a period of predefined length. We assume that the *get* and *inform*

operations consume negligible time (unless *get* finds no pending task, in which case it blocks, but returns immediately when a new task is injected). The computation part of the cycle, which involves executing a task, consumes the time needed for the specific task to be computed divided by the *speedup* $s \geq 1$. Processor cycles may not complete: An algorithm may decide to break the current cycle of a processor at any moment, in which case the processor starts a new one. Similarly, a crash failure breaks (forcefully) the cycle of a processor. Then, when the processor restarts, a new cycle begins.

Work conserving. We consider all *online* algorithms to be *work conserving*; not to allow any processor to idle when there are pending tasks and never break a cycle.

Event ordering. Due to the concurrent nature of the assumed computing system, processors' cycles may overlap between themselves and with the clients' inject operations. We therefore specify the following event ordering *at the repository* at a time t: first, the *inform* operations executed by processors are processed, then the *inject* operations, and last the *get* operations of processors. This implies that the set of pending tasks returned by a *get* operation executed at time t includes, besides the older unperformed tasks, the tasks injected at time t, and excludes the tasks reported as performed at time t. (This event ordering is done only for the ease of presentation and reasoning; it does not affect the generality of results.)

Tasks. Each task is associated with a unique *identifier*, an *arrival time* (the time it was injected in the system based on the repository's clock), and a *cost*, measured as the time needed to be performed (without a speedup). Let c_{min} and c_{max} denote the smallest and largest, respectively, costs that tasks may have (unless otherwise stated, this information is known to the processors). Throughout the paper we refer to a task of cost $c \in [c_{min}, c_{max}]$, as a c-task. We assume that tasks are *atomic* with respect to their completion: if a processor stops executing a task (intentionally or due to a crash) before completing the entire task, then no partial information can be shared with the repository, nor the processor may resume the execution of the task from the point it stopped (i.e., preemption is not allowed). Note also, that if a processor performs a task but crashes before the *inform* operation, then this task is not considered completed. Finally, tasks are assumed to be *similar* (require equal or comparable resources), *independent*, and *idempotent* (multiple executions of the same task produce the same final result). Several applications involving tasks with such properties are discussed in [18].

Adversary. We assume an omniscient adversary that can cause processor crashes and restarts, as well as task injections (at the repository). We define an *adversarial pattern* $\mathcal{A}$ as a collection of crash, restart and injection events caused by the adversary. Each event is associated with the time it occurs (e.g., $crash(t, i)$ specifies that processor i is crashed at time t). We say that a processor i is *alive* in time interval $[t, t']$, if the processor is operational at time t and does not crash by time t'. We assume that a restarted processor has knowledge of only the algorithm being executed and parameter n (number of processors). Thus, upon a restart, a processor simply starts a new cycle.

Efficiency Measures. We evaluate our algorithms using the *pending cost* measure, defined as follows. Given a time point $t \geq 0$ of the execution of an algorithm ALG under an adversarial pattern $\mathcal{A}$, we define the *pending cost at time* t, $\mathcal{C}_t(\text{ALG}, \mathcal{A})$, to be the sum of the costs of the pending tasks at the repository at time t. Furthermore, we denote

the number of pending tasks at the repository at time t under adversarial pattern $\mathcal{A}$ by $\mathcal{T}_t(\text{ALG}, \mathcal{A})$.

Since we view the task performance problem as an online problem, we pursue competitive analysis. Specifically, we say that an algorithm ALG is *x-pending-cost competitive* if $\mathcal{C}_t(\text{ALG}, \mathcal{A}) \leq x \cdot \mathcal{C}_t(\text{OPT}, \mathcal{A}) + \Delta$, for any t and under any adversarial pattern $\mathcal{A}$; Δ can be any expression independent of $\mathcal{A}$ and $\mathcal{C}_t(\text{OPT}, \mathcal{A})$ is the minimum (or infimum, in case of infinite computations) pending cost achieved by any *off-line algorithm* —that knows a priori $\mathcal{A}$ and has unlimited computational power— at time t of its execution under the adversarial pattern $\mathcal{A}$. Similarly, we say that an algorithm ALG is *x-pending-task competitive* if $\mathcal{T}_t(\text{ALG}, \mathcal{A}) \leq x \cdot \mathcal{T}_t(\text{OPT}, \mathcal{A}) + \Delta$, where $\mathcal{T}_t(\text{OPT}, \mathcal{A})$ is analogous to $\mathcal{C}_t(\text{OPT}, \mathcal{A})$. We omit $\mathcal{A}$ from the above notations when it can be inferred from the context.

3 NP-hardness

We now show that the offline problem of optimally scheduling tasks to minimize pending cost or number of pending tasks is NP-hard. This justifies the approach used in this paper for the online problem, speeding up the processors. In fact we show NP-hardness for problems with even one single processor.

Let us consider $C\_SCHED(t, \mathcal{A})$ which is the problem of scheduling tasks so that the pending cost at time t under adversarial pattern $\mathcal{A}$ is minimized. We consider a decision version of the problem, $DEC\_C\_SCHED(t, \mathcal{A}, \omega)$, with an additional input parameter ω. An algorithm solving the decision problem outputs a Boolean value $TRUE$ if and only if there is a schedule that achieves pending cost no more than ω at time t under adversarial pattern $\mathcal{A}$. I.e., $DEC\_C\_SCHED(t, \mathcal{A}, \omega)$ outputs $TRUE$ if and only if $\mathcal{C}_t(\text{OPT}, \mathcal{A}) \leq \omega$.

Theorem 1. *The problem $DEC\_C\_SCHED(t, \mathcal{A}, \omega)$ is NP-hard.*

A similar theorem can be stated (and proved following the same line), for a decision version of a respective problem, say $DEC\_T\_SCHED(t, \mathcal{A})$ of $T\_SCHED(t, \mathcal{A}, \omega)$, for which the parameter to be minimized is the number of pending tasks.

4 Conditions on Non-Competitiveness

For given task costs c_{min}, c_{max} and speedup s, we define parameter γ as the smallest number (non-negative integer) of c_{min}-tasks that one processor can complete in addition to a c_{max}-task, such that no algorithm running without speedup can complete more tasks in the same time. The following properties are therefore satisfied:

Property 1. $\frac{\gamma c_{min} + c_{max}}{s} \leq (\gamma + 1)c_{min}$.
Property 2. For every non-negative integer $\kappa < \gamma$, $\frac{\kappa c_{min} + c_{max}}{s} > (\kappa + 1)c_{min}$.

It is not hard to derive that $\gamma = \max\{\lceil \frac{c_{max} - s c_{min}}{(s-1)c_{min}} \rceil, 0\}$.

We now present and prove necessary conditions for the speedup value to achieve competitiveness.

Theorem 2. *For any given c_{min}, c_{max} and s, if the following two conditions are satisfied*

(a) $s < \frac{c_{max}}{c_{min}}$*, and* ***(b)*** $s < \frac{\gamma c_{min} + c_{max}}{c_{max}}$

then no deterministic algorithm is competitive when run with speedup s against an adversary injecting tasks with cost in $[c_{min}, c_{max}]$ even in a system with one single processor.

In other words, if $s < \min\left\{\frac{c_{max}}{c_{min}}, \frac{\gamma c_{min} + c_{max}}{c_{max}}\right\}$ there is no deterministic competitive algorithm.

Proof. Consider a deterministic algorithm ALG. We define a universal off-line algorithm OFF with associated crash and injection adversarial patterns, and prove that the cost of OFF is always bounded while the cost of ALG is unbounded during the executions of these two algorithms under the defined adversarial crash-injection pattern.

In particular, consider an adversary that activates, and later keeps crashing and restarting one processor. The adversarial pattern and the algorithm OFF are defined recursively in consecutive *phases*, where formally each phase is a closed time interval and every two consecutive phases share an end. In each phase, the processor is restarted in the beginning and crashed at the end of the phase, while kept continuously alive during the phase. At the beginning of phase 1, there are γ of c_{min}-tasks and one c_{max}-task injected, and the processor is activated.

Suppose that we have already defined adversarial pattern and algorithm OFF till the beginning of phase $i \geq 1$. Suppose also, that during the execution of ALG there are x of c_{min}-tasks and y of c_{max}-tasks pending. The adversary does not inject any tasks until the end of the phase. Under this assumption we could simulate the choices of ALG during the phase i. There are two cases to consider:

Scenario 1. ALG schedules κ of c_{min}-tasks, where $0 \leq \kappa < \gamma$, and then schedules a c_{max}-task; then OFF runs $\kappa + 1$ of c_{min}-tasks in the phase, and after that the processor is crashed and the phase is finished. At the end, $\kappa + 1$ c_{min}-tasks are injected.

Scenario 2. ALG schedules $\kappa = \gamma$ of c_{min}-tasks; then OFF runs a single c_{max}-task in the phase, and after that the processor is crashed and the phase is finished. At the end, one c_{max}-task is injected.

What remains to show is that the definitions of the OFF algorithm and the associated adversarial pattern are valid, and that in the execution of OFF the number of pending tasks is bounded, while in the corresponding execution of ALG it is not bounded. Since the tasks have bounded cost, the same applies to the pending cost of both OFF and ALG. Here we give some useful properties of the considered executions of algorithms ALG and OFF, whose proofs can be found in [8].

Lemma 1. *The phases, adversarial pattern and algorithm OFF are well-defined. Moreover, in the beginning of each phase, there are exactly γ of c_{min}-tasks and one c_{max}-task pending in the execution of OFF.*

Lemma 2. *There are infinite number of phases.*

Lemma 3. *ALG never performs any c_{max}-task.*

Lemma 4. *If Scenario 2 was applied in the specification of a phase i, then the number of pending c_{max}-tasks at the end of phase i in the execution of ALG increases by one comparing with the beginning of phase i, while the number of pending c_{max}-tasks stays the same in the execution of OFF.*

Now we resume the main proof of non competitiveness, i.e., Theorem 2. By Lemma 1, the adversarial pattern and the corresponding offline algorithm OFF are well-defined and by Lemma 2, the number of phases is infinite. There are therefore two cases to consider: (1) If the number of phases for which Scenario 2 was applied in the definition is infinite, then by Lemma 4 the number of pending c_{max}-tasks increases by one infinitely many times, while by Lemma 3 it never decreases. Hence it is unbounded. (2) Otherwise (i.e., if the number of phases for which Scenario 2 was applied in the definition is bounded), after the last Scenario 2 phase in the execution of ALG, there are only phases in which Scenario 1 is applied, and there are infinitely many of them. In each such phase, ALG performs only κ of c_{min}-tasks while $\kappa + 1$ c_{min}-tasks will be injected at the end of the phase, for some corresponding non-negative integer $\kappa < \gamma$ defined in the specification of Scenario 1 for this phase. Indeed, the length of the phase is $(\kappa + 1)c_{min}$, while after performing κ of c_{min}-tasks ALG schedules a c_{max}-task and the processor is crashed before completing it, because $\frac{\kappa c_{min} + c_{max}}{s} > (\kappa + 1)c_{min}$ (cf., Property 2). Therefore, in every such phase of the execution of ALG the number of pending c_{min}-tasks increases by one, and it does not decrease since there are no other kinds of phases (recall that we consider phases with Scenario 1 after the last phase with Scenario 2 finished). Hence the number of c_{min}-tasks grows unboundedly in the execution of ALG.

To conclude, in both cases above, the number of pending tasks in the execution of ALG grows unboundedly in time, while the number of pending tasks in the corresponding execution of OFF (for the same adversarial pattern) is always bounded, by Lemma 1. ■

Note that the use of condition (a) is implicit in our proof.

5 Algorithm (n, β)-LIS

In this section we present Algorithm (n, β)-LIS, which balances between the following two paradigms: scheduling Longest-In-System task first (LIS) and redundancy avoidance. More precisely, the algorithm at a processor tries to schedule the task that has been waiting the longest and does not cause redundancy of work if the number of pending tasks is sufficiently large. See the algorithm pseudocode for details.

Algorithm (n, β)-LIS (for processor p)

Repeat *//Upon awaking or restart, start here*
  Get from the Repository the set of pending tasks $Pending$;
  Sort $Pending$ by task arrival and ids/costs; *//Ranking starts from* 0
  If $|Pending| \geq 1$
    then perform task with rank $p \cdot \beta n \mod |Pending|$;
  Inform the Repository of the task performed.

Recall that due to processes crashes, it would not be helpful for a process to notify the repository of the task it has scheduled before performing the task. Observe that since $s \geq c_{max}/c_{min}$, Algorithm (n, β)-LIS is able to complete one task for each task completed by the offline algorithm. Additionally, if there are at least βn^2 tasks pending, for $\beta \geq \frac{c_{max}}{c_{min}}$, two processors do not schedule the same task. Combining these two observations it is possible to prove that (n, β)-LIS is 1-task-competitive.

Theorem 3. *$\mathcal{T}_t((n,\beta)$-LIS$, \mathcal{A}) \leq \mathcal{T}_t(OPT, \mathcal{A}) + \beta n^2 + 3n$ and $\mathcal{C}_t((n,\beta)$-LIS$, \mathcal{A}) \leq \frac{c_{max}}{c_{min}} \cdot \big(\mathcal{C}_t(OPT, \mathcal{A}) + \beta n^2 + 3n\big)$, for any time t and adversarial pattern $\mathcal{A}$, and for speedup $s \geq \frac{c_{max}}{c_{min}}$, when $\beta \geq \frac{c_{max}}{c_{min}}$.*

Proof. We first focus on the number of pending-tasks. Suppose that (n, β)-LIS is not OPT $+ \beta n^2 + 3n$ competitive in terms of the number of pending tasks, OPT, for some $\beta \geq \frac{c_{max}}{c_{min}}$ and some $s \geq \frac{c_{max}}{c_{min}}$. Consider an execution witnessing this fact and fix the adversarial pattern associated with it together with the optimum solution OPT for it.

Let t^* be a time in the execution when $\mathcal{T}_{t^*}((n, \beta)\text{-LIS}) > \mathcal{T}_{t^*}(\text{OPT}) + \beta n^2 + 3n$. For any time interval I, let $\mathcal{T}_I$ be the total number of tasks injected in the interval I. Let $t_* \leq t^*$ be the smallest time such that for all $t \in [t_*, t^*)$, $\mathcal{T}_t((n, \beta)\text{-LIS}) > \mathcal{T}_t(\text{OPT}) + \beta n^2$ (Note that the selection of minimum time satisfying some properties defined by the computation is possible due to the fact that the computation is split into discrete processor cycles.) Observe that $\mathcal{T}_{t_*}((n, \beta)\text{-LIS}) \leq \mathcal{T}_{t_*}(\text{OPT}) + \beta n^2 + n$, because at time t_* no more than n tasks could be reported to the repository by OPT, while just before t_* the difference between (n, β)-LIS and OPT was at most βn^2.

Then, we have the following property.

Claim. $\mathcal{T}_{t^*}((n, \beta)\text{-LIS}) \leq \mathcal{T}_{t^*}(\text{OPT}) + \beta n^2 + 3n$.

The competitiveness for the number of pending tasks follows directly from the above claim: it violates the contradictory assumptions made in the beginning of the analysis. The result for the pending cost is a direct consequence of the one for pending tasks, as the cost of any pending task in (n, β)-LIS is at most $\frac{c_{max}}{c_{min}}$ times bigger than the cost of any pending task in OPT. ∎

6 Algorithm γn-Burst

Observe that, against an adversarial strategy where at first only one c_{max}-task is injected, and then only c_{min}-tasks are injected, algorithm (n, β)-LIS with one processor has unbounded competitiveness when $s < \frac{c_{max}}{c_{min}}$ (this can be generalized for n processors). This is also the case for algorithms using many other scheduling policies, e.g., ones that schedule first the more costly tasks. This suggests that for $s < \frac{c_{max}}{c_{min}}$ a scheduling policy that alternates executions of lower-cost and higher-cost tasks should be devised. In this section, we show that if the speed-up satisfies $\frac{\gamma c_{min} + c_{max}}{c_{max}} \leq s < \frac{c_{max}}{c_{min}}$ and the tasks can have only two different costs, c_{min} and c_{max}, then there is an algorithm, call it γn-Burst, that achieves 1-pending-task and 1-pending-cost competitiveness in a system with n processors. The algorithm's pseudocode follows.

We first overview the main idea behind the algorithm. Each processor groups the set of pending tasks into two sublists, L_{min} and L_{max}, each corresponding to the tasks

Algorithm γn-Burst (for processor p)

Input: c_{min}, c_{max}, n, s
Calculate $\gamma \leftarrow \lceil \frac{c_{max} - sc_{min}}{(s-1)c_{min}} \rceil$
Repeat *//Upon awaking or restart, start here*
  $c \leftarrow 0$; *//Reset the counter*
  Get from the Repository the set of pending tasks $Pending$;
  Create lists L_{min} and L_{max} of c_{min}- and c_{max}-tasks;
  Sort L_{min} and L_{max} according to task arrival;
  Case 1: $|L_{min}| < n^2$ and $|L_{max}| < n^2$
    If previously performed task was of cost c_{min} **then**
      perform task $(p \cdot n) \mod |L_{max}|$ in L_{max}; $c \leftarrow 0$; *//Reset the counter*
    else perform task $(p \cdot n) \mod |L_{min}|$ in L_{min}; $c \leftarrow \min(c+1, \gamma)$;
  Case 2: $|L_{min}| \geq n^2$ and $|L_{max}| < n^2$
    perform the task at position $p \cdot n$ in L_{min}; $c \leftarrow \min(c+1, \gamma)$;
  Case 3: $|L_{min}| < n^2$ and $|L_{max}| \geq n^2$
    perform the task at position $p \cdot n$ in L_{max}; $c \leftarrow 0$; *//Reset the counter*
  Case 4: $|L_{min}| \geq n^2$ and $|L_{max}| \geq n^2$
    If $c = \gamma$ **then** perform task at position $p \cdot n$ in L_{max}; $c \leftarrow 0$; *//Reset the counter*
    else perform task at position $p \cdot n$ in L_{min}; $c \leftarrow \min(c+1, \gamma)$;
  Inform the Repository of the task performed.

of cost c_{min} and c_{max}, respectively, ordered by arrival time. Following the same idea behind Algorithm (n, β)-LIS, the algorithm avoids redundancy when "enough" tasks are pending. Furthermore, the algorithm needs to take into consideration parameter γ and the bounds on speed-up s. For example, in the case that there exist enough c_{min}- and c_{max}-tasks (more than n^2 to be exact) each processor performs no more than γ consecutive c_{min}-tasks and then performs a c_{max}-task; this is the time it takes for the same processor to perform a c_{max}-task in OPT. To this respect, a counter is used to keep track of the number of consecutive c_{min}-tasks, which is *reset* when a c_{max}-task is performed. Special care needs to be taken for all other cases, e.g., when there are more than n^2 c_{max}-tasks pending but less than c_{min}-tasks, etc.

Theorem 4. *$\mathcal{T}_t(\gamma$n-Burst$, \mathcal{A}) \leq \mathcal{T}_t(OPT, \mathcal{A}) + 2n^2 + (3 + \left\lceil \frac{c_{max}}{s \cdot c_{min}} \right\rceil)n$, for any time t and adversarial pattern $\mathcal{A}$.*

The difference in the number of c_{max}-tasks between ALG and OPT can be shown to be bounded by $n^2 + 2n$. This, and Theorem 4, yield the following bound on the pending cost of γn-Burst, which also implies that it is 1-pending-cost competitive.

Theorem 5. *$\mathcal{C}_t(\gamma$n-Burst$, \mathcal{A}) \leq \mathcal{C}_t(OPT, \mathcal{A}) + c_{max}(n^2 + 2n) + c_{min}(n^2 + (1 + \left\lceil \frac{c_{max}}{s \cdot c_{min}} \right\rceil)n)$, for any time t and adversarial pattern $\mathcal{A}$.*

7 Algorithm LAF

In the case of only two different costs, we can obtain a competitive solution for speedup that matches the lower bound from Theorem 2. More precisely, for given two different cost values, c_{min} and c_{max}, we can compute the minimum speedup s^* satisfying condition (b) from Theorem 2 for these two costs, and choose (n, β)-LIS with speedup

c_{max}/c_{min} in case $c_{max}/c_{min} \leq s^*$ and γn-Burst with speedup s^* otherwise. (Note that s^* is upper bounded by 2.) However, in the case of more than two different task costs we cannot use γn-Burst, and so far we could only rely on (n, β)-LIS with speedup c_{max}/c_{min}, which can be large.

We would like to design a "substitute" for algorithm γn-Burst, working for any bounded number of different task costs, which is competitive for some fixed small speedup. (Note that $s \geq 2$ is enough to guarantee that condition (b) does not hold.) This algorithm would be used when c_{max}/c_{min} is large. In this section we design such an algorithm, that works for any bounded number of different task costs, and is competitive for speedup $s \geq 7/2$. This algorithm, together with algorithm (n, β)-LIS, guarantee competitiveness for speedup $s \geq \min\{\frac{c_{max}}{c_{min}}, 7/2\}$. In more detail, one could apply (n, β)-LIS with speedup $\frac{c_{max}}{c_{min}}$ when $\frac{c_{max}}{c_{min}} \leq 7/2$ and the new algorithm with speedup $7/2$ otherwise.

We call the new algorithm *Largest_Amortized_Fit* or LAF for short. It is parametrized by $\beta \geq c_{max}/c_{min}$. This algorithm is more "geared" towards pending cost efficiency. In particular, each processor keeps the variable $total$, storing the total cost of tasks reported by processor p, since the last restart (recall that upon a restart processors have no recollection of the past). For every possible task cost, pending tasks of that cost are sorted using the Longest-in-System (LIS) policy. Each processor schedules the largest cost task which is not bigger than $total$ and is such, that the list of pending tasks of the same cost (as the one selected) has at least βn^2 elements, for $\beta \geq c_{max}/c_{min}$. If there is no such task then the processor schedules an arbitrary pending one.

As we show in the full version [8], in order for the algorithm to be competitive, the number of different costs of injected tasks must be finite in the range $[c_{min}, c_{max}]$. Otherwise, the number of tasks of the same cost might never be larger than βn^2, which is necessary to assure redundancy avoidance. Whenever this redundancy avoidance is possible, the algorithm behaves in a conservative way in the sense that it schedules a large task, but not larger than the total cost already completed. This implies that in every life period of a processor (the continuous period between a restart and a crash of the processor) only a constant fraction of this period could be wasted (wrt the total task cost covered by OPT in the same period). Based on this observation, a non-trivial argument shows that a constant speedup suffices for obtaining 1-pending-cost competitiveness.

Theorem 6. *Algorithm LAF is 1-pending-cost competitive, and thus $\frac{c_{max}}{c_{min}}$-pending-task competitive, for speedup $s \geq 7/2$, provided the number of different costs of tasks in the execution is finite.*

8 Conclusions

In this paper we have shown that a speedup $s \geq \min\left\{\frac{c_{max}}{c_{min}}, \frac{\gamma c_{min}+c_{max}}{c_{max}}\right\}$ is *necessary* and *sufficient* for competitiveness.

One could argue that the algorithms we propose assume the knowledge of c_{min} and c_{max}, which may seem unrealistic. However, in practice, processors can estimate the smallest and largest task costs from the costs seen so far, and use these values as c_{min} and c_{max} in the algorithms. This results in a similar performance (up to constant

factors) of the proposed algorithms with this adaptive computation of c_{min} and c_{max} , with some minor changes in the analysis.

A research line that we believe worth of further investigation is to study systems where processors could use different speedups or their speedup could vary over time or even to accommodate dependent tasks.

References

1. Enhanced intel speedstep technology for the intel pentium m processor. Intel White Paper 301170-001, 2004.
2. M. Ajtai, J. Aspnes, C. Dwork, and O. Waarts. A theory of competitive analysis for distributed algorithms. In *Proceedings of the 35th Symposium on Foundations of Computer Science (FOCS 1994)*, pages 401–411, 1994.
3. S. Albers and A. Antoniadis. Race to idle: New algorithms for speed scaling with a sleep state. In *Proceedings of the 23rd ACM-SIAM Symposium on Discrete Algorithms (SODA 2012*, pages 1266–1285, 2012.
4. S. Albers, A. Antoniadis, and G. Greiner. On multi-processor speed scaling with migration. In *Proceedings of the 23rd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA 2011)*, pages 279–288, 2011.
5. D. Alistarh, M. A. Bender, S. Gilbert, and R. Guerraoui. How to allocate tasks asynchronously. In *Proceedings of the 53rd IEEE Symposium on Foundations of Computer Science (FOCS 2012)*, pages 331–340, 2012.
6. S. Anand, N. Garg, and N. Megow. Meeting deadlines: How much speed suffices? In *Proceedings of the 38th International Colloquium on Automata, Languages and Programming (ICALP 2011)*, pages 232–243, 2011.
7. R.J. Anderson and H. Woll. Algorithms for the certified Write-All problem. *SIAM Journal of Computing*, 26(5):1277–1283, 1997.
8. A. Fernandez Anta, Ch. Georgiou, D. R. Kowalski, and E. Zavou. Online parallel scheduling of non-uniform tasks: Trading failures with energy. In ArXiv, 2013.
9. B. Awerbuch, S. Kutten, and D. Peleg. Competitive distributed job scheduling. In *Proceedings of the 24th ACM Symposium on Theory of Computing (STOC 1992)*, pages 571–580, 1992.
10. N. Bansal, H. L. Chan, and K. Pruhs. Speed scaling with an arbitrary power function. In *Proceedings of the 20th ACM-SIAM Symposium on Discrete Algorithms (SODA 2009)*, pages 693–701, 2009.
11. H. L. Chan, J. Edmonds, and K. Pruhs. Speed scaling of processes with arbitrary speedup curves on a multiprocessor. In *Proceedings of the 21st ACM Symposium on Parallelism in Algorithms and Architectures (SPAA 2009)*, pages 1–10, 2009.
12. B. Chlebus, R. De-Prisco, and A.A. Shvartsman. Performing tasks on restartable message-passing processors. *Distributed Computing*, 14(1):49–64, 2001.
13. G. Cordasco, G. Malewicz, and A. Rosenberg. Advances in IC-Scheduling theory: Scheduling expansive and reductive dags and scheduling dags via duality. *IEEE Transactions on Parallel and Distributed Systems*, 18(11):1607–1617, 2007.
14. J. Dias, E. Ogasawara, D. de Oliveira, E. Pacitti, and M. Mattoso. A lightweight execution framework for massive independent tasks. In *Proceedings of the 3rd IEEE Workshop on Many-Task Computing on Grids and Supercomputers*, 2010.
15. Y. Emek, M. M. Halldorsson, Y. Mansour, B. Patt-Shamir, J. Radhakrishnan, and D. Rawitz. Online set packing and competitive scheduling of multi-part tasks. In *Proceedings of the 29th ACM Symposium on Principles of Distributed Computing (PODC 2010)*, page 2010, 440–449.

16. Enabling Grids for E-sciencE (EGEE). http://www.eu-egee.org.
17. Ch. Georgiou and D. R. Kowalski. Performing dynamically injected tasks on processes prone to crashes and restarts. In *Proceedings of the 25th International Symposium on Distributed Computing, (DISC 2011)*, pages 165–180. Springer, 2011.
18. Ch. Georgiou and A. A. Shvartsman. *Do-All Computing in Distributed Systems: Cooperation in the Presence of Adversity*. Springer, 2008.
19. G. Greiner, T. Nonner, and A. Souza. The bell is ringing in speed-scaled multiprocessor scheduling. In *Proceedings of the 21st ACM Symposium on Parallelism in Algorithms and Architectures (SPAA 2009)*, pages 11–18, 2009.
20. K.S. Hong and J.Y.T. Leung. On-line scheduling of real-time tasks. *IEEE Transactions on Computers*, 41(10):1326–1331, 1992.
21. K. Jeffay, D.F. Stanat, and C.U. Martel. On non-preemptive scheduling of period and sporadic tasks. In *Proceedings of the 12th Real-Time Systems Symposium*, pages 129–139, 1991.
22. P.C. Kanellakis and A.A. Shvartsman. *Fault-Tolerant Parallel Computation*. Kluwer Academic Publishers, 1997.
23. E. Korpela, D. Werthimer, D. Anderson, J. Cobb, and M. Lebofsky. Seti@home: Massively distributed computing for seti. *Computing in Science and Engineering*, 3(1):78–83, 2001.
24. Ph. Lalanda. Shared repository pattern. In *Proceedings of the 5th Pattern Languages of Programs Conference (PLoP 1998)*, 1998.
25. C. A. Phillips, C. Stein, E. Torng, and J. Wein. Optimal time-critical scheduling via resource augmentation. *Algorithmica*, 32(2):163–200, 2002.
26. M. L. Pinedo. *Scheduling: Theory, Algorithms, and Systems*. Springer, fourth edition, 2012.
27. K. Schwan and H. Zhou. Dynamic scheduling of hard real-time tasks and real-time threads. *IEEE Trans. Software Eng.*, 18(8):736–748, 1992.
28. D. Sleator and R. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28(2):202–208, 1985.
29. U. van Heesch, S. M. Hezavehi, and P. Avgeriou. Combining architectural patterns and software technologies in one design language. In *Proceedings of the 16th European Pattern Languages of Programming (EuroPLoP 2011)*, 2011.
30. A. Wierman, L.L.H. Andrew, and A. Tang. Power-aware speed scaling in processor sharing systems. In *Proceedings of IEEE INFOCOM 2009*, pages 2007–2015, 2009.
31. F. F. Yao, A. J. Demers, and S. Shenker. A scheduling model for reduced CPU energy. In *Proceedings of the 36th IEEE Symposium on Foundations of Computer Science (FOCS 1995)*, pages 374–382, 1995.
32. M. Yokokawa, F. Shoji, A. Uno, M. Kurokawa, and T. Watanabe. The k computer: Japanese next-generation supercomputer development project. In *Proceedings of the 2011 International Symposium on Low Power Electronics and Design (ISLPED 2011)*, pages 371–372, 2011.