Online Parallel Scheduling of Non-uniform Tasks: Trading Failures for Energy[☆]

Antonio Fernández Anta[a], Chryssis Georgiou[b], Dariusz R. Kowalski[c], Elli Zavou[a,d,1]

[a]*Institute IMDEA Networks*
[b]*University of Cyprus*
[c]*University of Liverpool*
[d]*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 machines 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 off-line optimal solution, even with one single machine and tasks of only two different execution times. We find that when additional energy is provided to the system in the form of processing 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 [35] and Internet-based supercomputing platforms such as SETI@home [26] and EGEE Grid [15] 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.

Much research has been dedicated to task scheduling problems over the last decades, each work addressing different challenges (e.g., [8, 11, 12, 13, 14, 16, 18, 19, 21, 24, 29, 34]). For example, many works address the issue of dynamic task injections, but do not consider failures (e.g., [10, 22]). Other works consider scheduling on one machine (e.g., [3, 30, 33]), with the drawback that the power of parallelism is not exploited (provided that tasks are independent). Some works consider failures, but assume that tasks are known a priori and their number is bounded (e.g., [5, 7, 11, 18, 19, 23, 24]), where others assume that tasks are uniform, that is, they have the same processing times (e.g., [16, 17]). Several works consider power-preserving issues, but do not consider, for example, failures (e.g., [9, 10, 34]).

[☆]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. A preliminary version of this work appears in the proceedings of FCT 2013.

[1]Partially supported by FPU Grant from MECD

Term	Description
$m \in \mathbb{N}$	Number of machines in the system
$s \geq 1$	Machine's speedup
c_{min}	Smallest task cost
c_{max}	Largest task cost
c-task	Task of cost $c \in [c_{min}, c_{max}]$
$\rho = \frac{c_{max}}{c_{min}}$	Cost ratio
$\gamma = \max\{\lceil \frac{\rho - s}{s-1} \rceil, 0\}$	Parameter γ used to define competitiveness thresholds
β	Parameter used for redundancy avoidance
Condition C1	$s < \rho$
Condition C2	$s < 1 + \gamma/\rho$

Table 1: Important notation and definitions

Contributions. In this work we consider a computing system in which tasks of *different* execution times arrive *dynamically and continuously* and must be executed by a set of $m \in \mathbb{N}$ machines that are prone to *crashes and restarts*. Due to the dynamicity involved, we view this task-executing problem as an online problem and pursue competitive analysis [2, 31]. We explore the impact of parallelism, different task execution times and faulty environment, on the competitiveness of the online system considered. Efficiency is measured as the maximum *number of pending tasks* as well as the maximum *pending cost* over any point in the execution, where pending tasks are the ones that have been injected in the system but are not completed yet, and pending cost is the sum of their execution times. An algorithm is considered to be x-pending-task competitive, if under any adversarial pattern (for both task arrivals and machine crashes and restarts) its pending task complexity is at most x times larger than the pending task complexity of the offline optimal algorithm OPT, under the same adversarial pattern. This holds similarly for x-pending-cost competitiveness, taking into account the pending cost complexity of the algorithms.

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 processoring *speed scaling* [6, 4, 10] is applied in the form of a *speedup* above a certain threshold. A speedup $s \in \mathbb{R}^+$ means that a machine can complete a task s times faster than the task's system specified execution time (and therefore has a meaning only when $s \geq 1$). The use of a speedup is a form of resource augmentation [28] and impacts the *energy consumption* of the machine. As a matter of fact, the power consumed (i.e., the energy consumed per unit of time) to run a machine at a speed x grows superlinearly with x, and it is typically assumed to have a form of $P = x^\alpha$, for $\alpha > 1$ [1, 34]. Hence, a speedup s implies an additional factor of $s^{\alpha-1}$ in the power consumed (and hence energy consumed).

Our investigation aims at developing competitive online algorithms that require the smallest possible speedup. As a result, one of the main challenges 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 off-line algorithm that possesses complete knowledge of failures and task injections? (It is understood that there is nothing to investigate if the off-line solution makes use of speed-scaling as well).

We now summarize our contributions. Table 1 provides usefull notation and definitions, and Table 2 provides an overview of our main results.

Formalization of fault-tolerant distributed scheduling: In Section 2, we formalize an online task executing 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 off-line solutions: In Section 3, we show that an off-line simpler version of our problem is NP-hard, for both

Condition	Number of task costs	Task competitiveness	Cost competitiveness	Algorithm
C1 $\wedge$ C2	≥ 2	∞	∞	Any
$\neg$C1	Any	1	ρ	(m,β)-LIS
C1 $\wedge$ $\neg$C2	2	1	1	γm-Burst
$s \geq 7/2$	Finite	ρ	1	(m,β)-LAF

Table 2: Summary of results. We define $\gamma = \max\{\lceil \frac{\rho-s}{s-1} \rceil, 0\}$ to be a parameter representing the number of c_{min}-tasks that, in addition to a c_{max}-task, an algorithm with speedup s can complete in a time interval of length $(\gamma+1)c_{min}$. Parameter β is a parameter of Algorithms LIS and LAF, used for avoiding redundancy of task executions. Also note that $\min\{\rho, 1+\gamma/\rho\} < 2$; this follows from the definitions of γ and ρ, and from $s \geq 1$.

pending task and pending cost efficiency. In the version considered, there is no parallelism (one machine) and the information of all tasks as well as the machine 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 $\gamma \in \mathbb{N}$, which represents the smallest number of c_{min}-tasks that an algorithm with speedup s can complete in addition to a c_{max}-task, such that the off-line algorithm (with $s = 1$) 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 also define parameter $\rho = c_{max}/c_{min}$ to be the ratio of the two costs and use it throughout the analysis.

We propose two conditions:

C1: $s < \rho$, and

C2: $s < 1 + \gamma/\rho$

and show that if *both* of them hold, then *no* deterministic sequential or parallel algorithm is competitive when run with speedup s. Observe that satisfying condition C2 implies $\gamma > 0$ (since $s \geq 1$), which automatically means that condition C1 is also satisfied when $\rho > 1$.

Note that this result holds even if we only have a single machine, and therefore could be generalized for "stronger" models that use centralized or parallel scheduling of multiple machines.

Sufficient conditions for competitiveness: Then, we design two scheduling algorithms, matching the different threshold bounds from the necessary conditions above, showing that they are also *sufficient*, leading to competitive solutions.

Algorithm (m,β)-LIS: For the case when condition C1 does not hold (i.e., $s \geq \rho$), we develop algorithm (m,β)-LIS, presented in Section 5. We show that under these circumstances, (m,β)-LIS is 1-pending-task-competitive and ρ-pending-cost-competitive, for parameter $\beta \geq \rho$ and for any given number of machines m. These results hold for any collection of tasks with costs in the range $[c_{min}, c_{max}]$.

Algorithm γm-Burst: It is not difficult to observe that algorithm (m,β)-LIS cannot be competitive when condition C1 holds but condition C2 does not (i.e., $1+\gamma/\rho \leq s < \rho$). For this case we develop algorithm γm-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. In the case of two different task costs, establishing speedup $s = \min\{\rho, 1+\gamma/\rho\}$ suffices for achieving competitiveness. In Section 7 we show that it is sufficient to set $s = \rho$ if $\rho \in [1, \varphi]$, and $s = 1 + \sqrt{1-1/\rho}$ otherwise, where $\varphi = \frac{1+\sqrt{5}}{2}$ is the golden ratio.

Algorithm (m,β)-LAF, low energy guaranteed: In Section 8 we develop algorithm (m,β)-LAF that is again competitive for the case when condition C2 does not hold, but in contrast with γm-Burst, it is more "geared" towards pending cost efficiency and can handle tasks of multiple different costs. We show that unlike the above mentioned algorithms, this one is ρ-pending-task competitive and 1-pending-cost competitive for speedup $s \geq \frac{7}{2}$. Its importance lies on the fact that competitiveness is with respect to a speedup that is independent of the values c_{min} and c_{max}.

Task Scheduling. We assume the existence of an entity, called *Shared Repository* (whose detailed specification is given in Section 2). This entity abstracts the service by which clients submit computational tasks to our system and notifies them when they are completed. This allows our results to be conceptually general instead of considering specific implementation details. Since the Shared Repository does not make any task allocation decisions, it is *not* a scheduler. The machines access this entity only to obtain the set of pending tasks. An example of such an entity and implementations of it can be found in the Software Components Communication literature, where it is referred to as the Shared Repository Pattern (see for example [27, 32], and references therein).

The Shared Repository makes our setting simpler, easier to implement and more scalable than other popular settings with stronger scheduling computing entities, such as a *central scheduler*. Even in the case of the central scheduler, a central repository is still 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 are done by a single entity which allocates the tasks to the machines, whereas in our setting scheduling decisions are done in *parallel* by the participating machines. As a consequence, all the results of our work also hold for such stronger models: algorithms work at least as good as in the Shared Repository setting since it is a weaker model, and the necessary conditions on the energy (and thus speedup) threshold also hold as they are proven for a scenario with a single machine, where these two models are indistinguishable.

Another subtle issue between a setting where machines make scheduling decisions in parallel (as in our setting) and a setting with central scheduling, is that in the latter it is easier (algorithmically speaking) to impose *redundancy avoidance* (each task is executed by exactly one machine). Redundancy avoidance is an interesting issue by its own right. It is for example a prerequisite for the *At-Most-Once problem* [25], where given a set of tasks, none should be executed more than once by any machine. In our algorithms, we use a simple mechanism that imposes redundancy avoidance whenever there is a sufficiently large number of pending tasks. More sophisticated redundancy avoidance mechanisms could only improve on additive parts of the competitiveness formulas obtained in this work.

Related Work. The most closely related work to this one is the one by Georgiou and Kowalski [16]. As in this work, they consider a task-executing problem where tasks are dynamically and continuously injected in 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 completed 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 is what has essentially motivated 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 was not trivial; different scheduling policies and techniques had to be devised.

Our work is also related to studies of parallel online scheduling using identical machines [29]. Among them, several papers consider speed-scaling and speedup issues. Some of them, unlike our work, consider dynamic scaling (e.g., [4, 9, 10]). Usually, in these works preemption is allowed: an execution of a task may be suspended and later restarted from the point of suspension. However, in our work, the task must be executed from scratch. The authors of [20] 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 authors of work [6] consider 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.

Finally, Boyar et al. [23] have recently looked into the Grid Scheduling problem, which is closely related to our work. It can be seen as a bin packing problem for a set of items given from the beginning and bins of different sizes that arrive dynamically and have to eventually serve all the items in the set. One can see the correlation between their work and ours if (s)he relates the arrival of processors in the former with the length of periods that machines are alive in the latter. The authors perform competitive analysis and give lower and upper bounds as in our work. Nonetheless, there are some important differences from our work. For example, the set of jobs that have to be completed is finite and known from the beginning, whereas we consider a real-time dynamic arrival of tasks and their performance. Also, the competitive ratios they give depend on the number of jobs of each size, whereas our results depend only on the ratio of the costs of the largest and smallest tasks.

2. Model and Definitions

Computing Setting. We consider a system of m homogeneous, fault-prone machines, with unique ids from the set $[m] = \{1, 2, \ldots, m\}$. We assume that machines 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 completed 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. As discussed below, this operation is controlled by an adversary, whereas the other two operations are executed by the system's machines. By executing a *get* operation, a machine 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 of their completion 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 machine 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 the machine crashes, it would not be helpful for a machine to notify the repository simply when scheduling a task before it has actually executed it completely. Last, each operation performed by a machine 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).
Processing cycles. Machines 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 machine 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* $s \geq 1$. What is more, processing cycles may not complete; an algorithm may decide to break the current cycle of a machine at any moment, in which case the machine starts a new one. In a similar way, a crash failure breaks (forcefully) the processing cycle of a machine and when the machine restarts, a new cycle begins.
Event ordering. Due to the concurrent nature of the computing system considered, machine's processing 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 machines are processed, then the *inject* operations, and last the *get* operations of machines. This implies that the set of pending tasks returned by a *get* operation executed at time t includes, besides the older uncompleted tasks, the tasks injected at time t, and excludes the tasks reported as completed at time t.[2]
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 executed (without a speedup). Let c_{min} and c_{max} denote the smallest and largest costs that tasks may have respectively (unless otherwise stated, this information is known to the machines). 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 machine stops executing a task before completing it (intentionally or due to a crash), then neither any partial information can be shared with the repository, nor the machine may resume the execution of the task from the point it stopped (i.e., preemption is not allowed). Note also, that if a machine executes a task but crashes before the *inform* operation, then this task is not considered completed. Finally, all machines are *identical* (a task computation on any of them consumes equal or comparable local resources). Moreover,

[2] This event ordering is done only for the ease of presentation and reasoning; it does not affect the generality of results.

tasks are assumed to be *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 machine 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 machine i is crashed at time t). We say that a machine i is *alive* in time interval $[t, t']$, if the machine is operational at time t and does not crash by time t'. We assume that a restarted machine has knowledge only of the algorithm being executed and parameter m (number of machines). Thus, upon a restart, a machine simply starts a new processing cycle.

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

Since we view the task execution problem as an online problem, we pursue competitive analysis. Specifically, we say that an algorithm ALG is x*-pending-task competitive* if $\mathcal{T}_t(\text{ALG}, \mathcal{A}) \leq x \cdot \mathcal{T}_t(\text{OPT}, \mathcal{A}) + \Delta$, for any t and under any adversarial pattern $\mathcal{A}$. Δ can be any expression independent of t, $\mathcal{A}$, and $\mathcal{T}_t(\text{OPT}, \mathcal{A})$, but might however depend on the system parameters. These parameters, like c_{min}, c_{max}, m or s, are fixed and given upfront to the algorithms and hence are not part of the input of the problem, which is formed by the adversarial pattern only. In particular, it is important to clarify that the number of machines m is *fixed* for a given execution, and that the algorithm that tackles it may take it into consideration; hence different m may result to different performance of the same algorithm, due to the additive term in the competitiveness. $\mathcal{T}_t(\text{OPT}, \mathcal{A})$ is the minimum number (or infimum in case of infinite computations) of pending tasks achieved by any *off-line algorithm* —that knows a priori $\mathcal{A}$ and has unlimited computational power— at time t of its execution and under the adversarial pattern $\mathcal{A}$. Similary, we say that an algorithm ALG is x*-pending-cost competitive* if $\mathcal{C}_t(\text{ALG}, \mathcal{A}) \leq x \cdot \mathcal{C}_t(\text{OPT}, \mathcal{A}) + \Delta$, where $\mathcal{C}_t(\text{OPT}, \mathcal{A})$ is analogous to $\mathcal{T}_t(\text{OPT}, \mathcal{A})$. We omit $\mathcal{A}$ from the above notations when it can be inferred from the context.

3. NP-hardness

We now show that the off-line problem of optimally scheduling tasks in order to minimize the number of pending tasks or the pending cost is NP-hard. This justifies the approach used in this paper for the online problem; speeding up the machines. In fact we show NP-hardness for problems with even one single machine. This implies the NP-hardness of the problem with more machines as well, since the adversary could crash all but one machine.

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.*

Proof. The reduction we use is from the Partition problem. The input considered is a set of numbers (we assume positive) $C = \{x_1, x_2, ..., x_k\}$, $k > 1$. The problem is to decide whether then, there is a subset $C' \subset C$ such that $\sum_{x_i \in C'} x_i = \frac{1}{2} \sum_{x_i \in C} x_i$. The Partition problem is know to be NP-complete.

Consider any instance I_p of Partition. We construct an instance I_d of $DEC\_C\_SCHED(t, \mathcal{A}, \omega)$ as follows. The time t is set to $1 + \sum_{x_i \in C} x_i$. The adversarial pattern $\mathcal{A}$ injects a set S of k tasks at time 0, so that the ith task has cost x_i. It also starts the machine at time 0 and crashes it at time $\frac{1}{2} \sum_{x_i \in C} x_i$. Then, $\mathcal{A}$ restarts the machine immediately and crashes it again at time $\sum_{x_i \in C} x_i$. The machine does not restart until time t. Finally, the parameter ω is set to 0.

Assume there is an algorithm ALG that solves DEC_C_SCHED. We show that ALG can be used to solve the instance I_p of Partition by solving the instance I_d of DEC_C_SCHED obtained as described. If there is a $C' \subset C$ such that $\sum_{x_i \in C'} x_i = \frac{1}{2} \sum_{x_i \in C} x_i$, then there is an algorithm that is able to schedule tasks from S so that the two semi-periods (of length $\frac{1}{2} \sum_{x_i \in C} x_i$ each) the machine is active, it is doing useful work. In that case, the pending cost at time t will be $0 = \omega$. If, on the other hand, such subset does not exist, some of the time the machine is active will be wasted, and the cost pending at time t has to be larger than ω. $\square$

A similar theorem can be stated (and proved following the same line), for a decision version of a respective problem, say $DEC\_T\_SCHED(t, \mathcal{A}, \omega)$ of $T\_SCHED(t, \mathcal{A})$, 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 number of c_{min}-tasks, in addition to a c_{max}-task, that an algorithm with speedup s can complete in a time interval of length $(\gamma + 1)c_{min}$. The following properties are therefore satisfied:

Property 1. $\frac{\gamma c_{min} + c_{max}}{s} \le (\gamma + 1)c_{min}$.

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 $\gamma = \max\{\lceil \frac{c_{max} - s c_{min}}{(s-1)c_{min}} \rceil, 0\} = \max\{\lceil \frac{\rho - s}{s-1} \rceil, 0\}$ (recall that $\rho = c_{max}/c_{min}$). Observe that by the definitions of γ and ρ, and that $s \ge 1$, we have that $\min\{\rho, 1 + \gamma/\rho\} < 2$.

We now prove that both conditions C1 and C2 presented earlier (Table 1), are necessary for the value of speedup in order to achieve competitiveness. If both are satisfied, then no deterministic algorithm is competitive against an adversary injecting tasks with costs in the interval $[c_{min}, c_{max}]$, even in a system with one single machine. In other words, if $s < \min\{\rho, 1 + \gamma/\rho\}$ there is no deterministic competitive algorithm.

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 re-starting one machine. 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. The machine is restarted at the beginning and crashed at the end of each 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 machine is activated.

Suppose that we have already defined the adversarial pattern and algorithm OFF till the beginning of phase $i \ge 1$. Suppose also that in the execution of ALG there are x of c_{min}-tasks and y of c_{max}-tasks pending at the beginning of phase i. 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, illustrated in Figures 1 and 2. Let us first define parameter Δ to be the time elapsed from the beginning of phase i until the time at which ALG starts executing a c_{max}-task, with an intention to complete it (assuming phase i is long enough). Note here that since ALG is deterministic, the adversary knows the times at which ALG decides to stop any processing cycle and schedule another task. The two possible scenarios are therefore the following:

Scenario 1. When $\Delta < \gamma c_{min}/s$, ALG schedules a c_{max}-task sooner than $\gamma c_{min}/s$ time from the beginning of the phase. Let $\kappa = \lfloor \Delta/(c_{min}/s) \rfloor < \gamma$. The adversary ends the phase $(\kappa + 1)c_{min}$ time after the beginning of the phase. From Property 2, $\frac{\kappa c_{min} + c_{max}}{s} > (\kappa + 1)c_{min}$. Therefore, before the end of the phase, OFF has enough time to complete $\kappa + 1$ tasks of cost c_{min}, while ALG cannot complete more than κ. Moreover, ALG cannot complete the execution of the c_{max} task. At the end of the phase, the adversary injects $\kappa + 1$ tasks of cost c_{min}.

Scenario 2. When $\Delta \ge \gamma c_{min}/s$, ALG schedules a c_{max}-task no sooner than $\gamma c_{min}/s$ time after the phase starts. On the same time, OFF is able to run a c_{max}-task. The adversary crashes the machine when OFF completes the c_{max}-task, and the phase finishes. At the end of the phase, the adversary injects one c_{max}-task, as OFF has completed one. In this scenario, ALG is at most able to complete γ tasks of cost c_{min}.

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. We now give some useful properties of the considered executions of algorithms ALG and OFF, before completing the proof of the theorem.

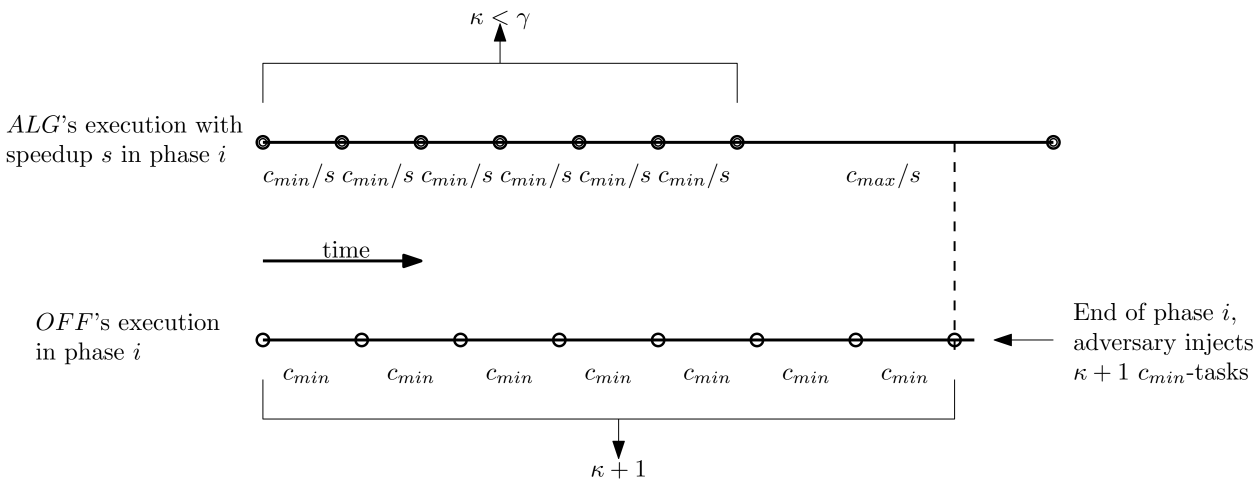
$\kappa < \gamma$
ALG's execution with speedup s in phase i
c_{min}/s c_{min}/s c_{min}/s c_{min}/s c_{min}/s c_{min}/s
c_{max}/s
time
OFF's execution in phase i
c_{min} c_{min} c_{min} c_{min} c_{min} c_{min} c_{min}
End of phase i, adversary injects $\kappa + 1$ c_{min}-tasks
$\kappa + 1$

Figure 1: Illustration of Scenario 1. It uses the property $(\kappa c_{min} + c_{max})/s > (\kappa + 1)c_{min}$, for any integer $0 \leq \kappa < \gamma$ (Property 2).

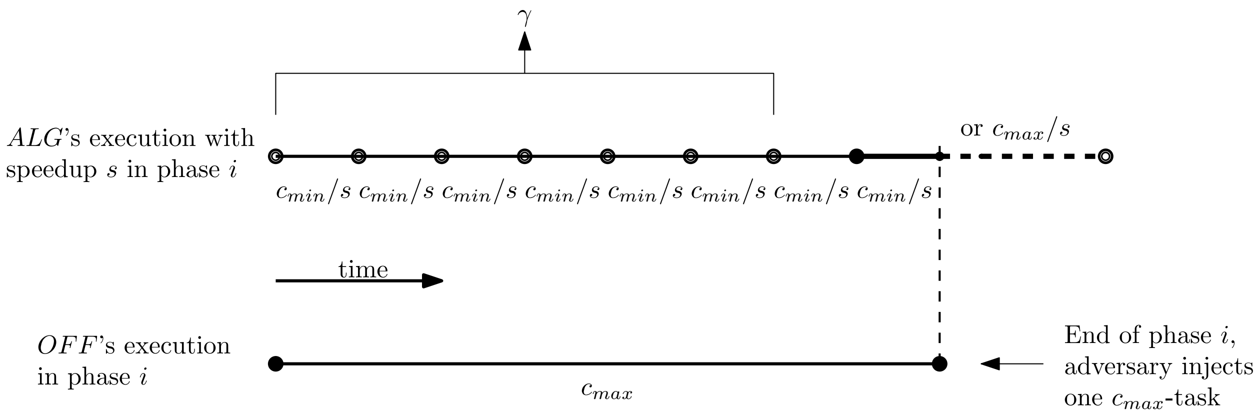
γ
ALG's execution with speedup s in phase i
or c_{max}/s
c_{min}/s c_{min}/s c_{min}/s c_{min}/s c_{min}/s c_{min}/s c_{min}/s c_{min}/s
time
OFF's execution in phase i
c_{max}
End of phase i, adversary injects one c_{max}-task

Figure 2: Illustration of Scenario 2. It uses the property $(\gamma c_{min} + c_{max})/s > c_{max}$ (condition C2).

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

Proof. We argue by induction on the number of phases that: at the beginning of phase i there are exactly γ of c_{min}-tasks and one c_{max}-task pending in the execution of OFF, and therefore phase i is well defined. Its specification (including termination time) depends only on whether OFF schedules either up to γ of c_{min}-tasks (in Scenario 1) or one c_{max}-task (in Scenario 2) before the next task injection at the end of the phase. The invariant holds for phase 1 by definition. By straightforward investigation of both scenarios, the very same configuration of task lengths that has been completed by OFF in its execution during a phase is injected at the end of the phase, and therefore the inductive argument proves the invariant for every consecutive phase. □

Lemma 2. *There is an infinite number of phases.*

Proof. First, by Lemma 1, consecutive phases are well-defined. Second, observe that each phase is finite, regardless of whether Scenario 1 or Scenario 2 is applied. It is bounded by the time ALG schedules a c_{max}-task which results to phases of size equal to the time needed by OFF to complete either at most γ of c_{min}-tasks (in Scenario 1) or exactly one c_{max}-task (in Scenario 2). Hence, in an infinite execution the number of phases is infinite. □

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

Proof. It follows from the specification of Scenarios 1 and 2, condition C2 on the speedup s, and from Property 2. Considering a phase, if Scenario 1 is applied for its specification, then ALG could not finish its c_{max}-task scheduled after $\kappa < \gamma$ tasks of cost c_{min}, because the time needed for completing this sequence of tasks is at least $\frac{\kappa c_{min}+c_{max}}{s}$, which is larger than the length of this phase $(\kappa+1)c_{min}$ (Property 2). If Scenario 2 is applied for the phase specification, then the first c_{max}-task could be finished by ALG no earlier than $\frac{\gamma c_{min}+c_{max}}{s}$ time after the beginning of the phase, which is again bigger than the length of the phase, c_{max} (by the assumption of condition C2 on the speedup $s < 1+\gamma/\rho = \frac{\gamma c_{min}+c_{max}}{c_{max}}$). □

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 the phase in the execution of ALG increases by one comparing with the beginning of the phase. In the execution of OFF on the other hand, the number of pending c_{max}-tasks stays the same.*

Proof. It follows from Lemma 3 and from the specification of tasks injections at the end of phase i, by Scenario 2. □

Putting everything together, we now prove Theorem 2.

Theorem 2. *For any given c_{min}, c_{max} and s, if both conditions* ***C1:*** *$s < \rho$, and* ***C2:*** *$s < 1+\gamma/\rho$ are satisfied, then no deterministic algorithm is competitive when run with speedup s against an adversary injecting tasks with costs in the interval $[c_{min}, c_{max}]$, even in a system with one single machine.*

Proof. By Lemma 1, the adversarial pattern and the corresponding off-line 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 completes only κ of c_{min}-tasks (where $\kappa < \gamma$) while $\kappa+1$ tasks of cost c_{min} will be injected at the end of the phase. Indeed, the length of the phase is $(\kappa+1)c_{min}$, while after completing κ of c_{min}-tasks ALG schedules a c_{max}-task and the machine 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.[3] □

5. Algorithm (m, β)-LIS

In this section we present Algorithm (m, β)-LIS, which balances between scheduling Longest-In-System task first (LIS) and redundancy avoidance. More precisely, the algorithm at a machine 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's pseudocode (Algorithm 1) for details and observe that since $s \geq \rho$, Algorithm (m, β)-LIS is able to complete one task for each task completed by the off-line algorithm. Additionally, if there are at least βm^2 tasks pending, where $\beta = \rho$, no two machines schedule the same task.

We show that algorithm (m, β)-LIS is 1-pending-task and ρ-pending-cost competitive for speedup $s \geq \rho$ when $\beta \geq \rho$. We first provide a high-level idea of the proof and then we proceed to rigorously prove the claimed result.

Overview of the proof: We focus on the number of pending tasks competitiveness, by which the result on the pending cost follows. We assume by contradiction, that (m, β)-LIS is not OPT + βm^2 + $3m$ competitive in terms of the

[3]Note that the use of condition C1 is implicit in our proof.

Algorithm 1 (m, β)**-LIS** (for machine p)

```
Parameters: $m, \beta$
Repeat                                                        //Upon awaking or restart
    Get from the Repository the set of pending tasks $Pending$;
    Sort $Pending$ by task arrival and ids/costs;
    If $|Pending| \geq 1$ then
        execute task with rank $p \cdot \beta m \mod |Pending|$;
    Inform the Repository of the task executed.
```

number of pending tasks, for some $\beta \geq \rho$ and some $s \geq \rho$. We consider an execution witnessing this fact and fix the adversarial pattern associated with it together with the optimum solution OPT for it.

Then, we define t^* be a time in the execution when $\mathcal{T}_{t^*}((m,\beta)\text{-LIS}) > \mathcal{T}_{t^*}(\text{OPT}) + \beta m^2 + 3m$ and let $t_* \leq t^*$ be the smallest time such that for all $t \in [t_*, t^*)$, $\mathcal{T}_t((m,\beta)\text{-LIS}) > \mathcal{T}_t(\text{OPT}) + \beta m^2$. 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 processing cycles. Also, observe that $\mathcal{T}_{t_*}((m,\beta)\text{-LIS}) \leq \mathcal{T}_{t_*}(\text{OPT}) + \beta m^2 + m$, because at time t_* no more than m tasks could be reported to the repository by OPT, while just before t_* the difference between (m,β)-LIS and OPT was at most βm^2.

We now use the above definitions to prove the following lemmas that will lead to contradiction of the initial assumption and yield the proof of the claimed result (Theorem 3).

Lemma 5. *We have $t_* < t^* - c_{min}$, and for every $t \in [t_*, t_* + c_{min}]$ the following holds with respect to the number of pending tasks: $\mathcal{T}_t((m,\beta)\text{-LIS}, \mathcal{A}) \leq \mathcal{T}_t(\text{OPT}, \mathcal{A}) + \beta m^2 + 2m$.*

Proof. We already discussed the case $t = t_*$. In the interval $(t_*, t_* + c_{min}]$, OPT can notify the repository about at most m completed tasks, as each of m machines may finish at most one task. Consider any $t \in (t_*, t_* + c_{min}]$ and let I be fixed to $(t_*, t]$. We have $\mathcal{T}_t((m,\beta)\text{-LIS}, \mathcal{A}) \leq \mathcal{T}_{t_*}((m,\beta)\text{-LIS}, \mathcal{A}) + \mathcal{T}_I$ and $\mathcal{T}_t(\text{OPT}, \mathcal{A}) \geq \mathcal{T}_{t_*}(\text{OPT}, \mathcal{A}) + \mathcal{T}_I - m$. It follows that

$$\begin{aligned}
\mathcal{T}_t((m,\beta)\text{-LIS}, \mathcal{A}) &\leq \mathcal{T}_{t_*}((m,\beta)\text{-LIS}, \mathcal{A}) + \mathcal{T}_I \\
&\leq \big(\mathcal{T}_{t_*}(\text{OPT}, \mathcal{A}) + \beta m^2 + m\big) + (\mathcal{T}_t(\text{OPT}, \mathcal{A}) - \mathcal{T}_{t_*}(\text{OPT}, \mathcal{A}) + m) \\
&\leq \mathcal{T}_t(\text{OPT}, \mathcal{A}) + \beta m^2 + 2m \ .
\end{aligned}$$

It also follows that any such t must be smaller than t^*, by definition of t^*. □

Lemma 6. *Consider a time interval I during which the queue of pending tasks in (m,β)-LIS is always non-empty. Then the total number of tasks reported by OPT in the period I is not bigger than the total number of tasks reported by (m,β)-LIS in the same period plus m (counting possible redundancy).*

Proof. For each machine in the execution of OPT, under the adversarial pattern $\mathcal{A}$, in the considered period, exclude the first reported task; this is to eliminate from further analysis tasks that might have been started before time interval I. There are at most m such tasks reported by OPT.

It remains to show that the number of remaining tasks reported to the repository by OPT is not bigger than those reported in the execution of (m,β)-LIS in the considered period I. It follows from the property that $s \geq \rho$, which implies that during the time period when a machine p executes a task τ in the execution of OPT, the same machine reports at least one task to the repository in the execution of (m,β)-LIS. This is because executing any task by a machine in the execution of OPT takes at least time c_{min}, while executing any task in the execution of (m,β)-LIS takes no more than $\frac{c_{max}}{s} \leq c_{min}$ (recall that $s \geq \rho = \frac{c_{max}}{c_{min}}$), and also because no active machine in the execution of (m,β)-LIS is ever idle (non-emptiness of the pending task queue). Hence we can define a 1-1 function from the considered tasks completed by OPT (i.e., tasks which are started and reported in time interval I) to the family of different tasks reported by (m,β)-LIS in the period I, which completes the proof. □

Lemma 7. *In the interval $(t_* + c_{min}, t^*]$ no task is reported twice to the repository by (m, β)-LIS.*

Proof. The proof is by contradiction. Suppose that task τ is reported twice in the considered time interval of the execution of (m, β)-LIS, under adversarial pattern $\mathcal{A}$. Consider the first two such reports, by machines p_1 and p_2; w.l.o.g. we may assume that p_1 reported τ at time t_1, not later than p_2 reported τ at time t_2. Let c_τ denote the cost of task τ. The considered reports have to occur within time period shorter than the cost of task τ, in particular, shorter than $c_{max}/s \leq c_{min}$; otherwise it would mean that the machine which reported second would have started executing this task not earlier than the previous report to the repository, which contradicts the property of the repository that each reported task is immediately removed from the list of pending tasks. It also implies that $p_1 \neq p_2$.

From the algorithm description, the list $Pending$ at time $t_1 - c_\tau/s$ had task τ at position $p_1\beta n$, while the list $Pending$ at time $t_2 - c_\tau/s$ had task τ at position $p_2\beta m$. Note that interval $[t_1 - c_\tau/s, t_2 - c_\tau/s]$ is included in $[t_*, t^*]$, and thus, by the definition of t_*, at any time of this interval there are at least βm^2 tasks in the list $Pending$.

There are two cases to consider. First, if $p_1 < p_2$, then because new tasks on list $Pending$ are appended to the end of the list, it will never happen that a task with rank $p_1\beta m$ would increase its rank in time, in particular, not to $p_2\beta m$. Second, if $p_1 > p_2$, then during time interval $[t_1 - c_\tau/s, t_2 - c_\tau/s]$ task τ has to decrease its rank from $p_1\beta m$ to $p_2\beta m$, i.e., by at least βm positions. It may happen only if at least βm tasks ranked before τ on the list $Pending$ at time $t_1 - c_\tau/s$ become reported in the considered time interval. Since all of them are of cost at least c_{min}, and the considered time interval has length smaller than c_{max}/s, each machine may report at most $\frac{c_{max}/s}{c_{min}/s} \leq \beta$ tasks (this is the part of analysis requiring $\beta \geq \rho = \frac{c_{max}}{c_{min}}$). Since machine p_2 can report at most $\beta - 1$ tasks different than τ, the total number of tasks different from τ reported to the repository is at most $\beta m - 1$, and hence it is not possible to reduce the rank of τ from $p_1\beta m$ to $p_2\beta m$ within the considered time interval. This contradicts the assumption that p_2 reports τ to the repository at time t_2. □

Theorem 3. *For speedup $s \geq \rho$ when $\beta \geq \rho$ the following holds:*
$\mathcal{T}_t((m, \beta)$-LIS$, \mathcal{A}) \leq \mathcal{T}_t(OPT, \mathcal{A}) + \beta m^2 + 3m$ and $\mathcal{C}_t((m, \beta)$-LIS$, \mathcal{A}) \leq \rho \cdot \big(\mathcal{C}_t(OPT, \mathcal{A}) + \beta m^2 + 3m\big)$, for any time t and under adversarial pattern $\mathcal{A}$.

Proof. From Lemma 5, we know that $\mathcal{T}_{t_*+c_{min}}((m, \beta)\text{-LIS}, \mathcal{A}) \leq \mathcal{T}_{t_*+c_{min}}(\text{OPT}, \mathcal{A}) + \beta m^2 + 2m$. Now let y be the total number of tasks reported by (m, β)-LIS in $(t_* + c_{min}, t^*]$. By Lemma 6 and definitions t_* and t^*, OPT reports no more than $y + n$ tasks in $(t_* + c_{min}, t^*]$. Therefore,

$$\mathcal{T}_{t^*}(\text{OPT}, \mathcal{A}) \geq \mathcal{T}_{t_*+c_{min}}(\text{OPT}, \mathcal{A}) - (y + m)\ .$$

By Lemma 7, in the interval $(t_* + c_{min}, t^*]$, no redundant work is reported by (m, β)-LIS. Thus,

$$\mathcal{T}_{t^*}((m, \beta)\text{-LIS}, \mathcal{A}) \leq \mathcal{T}_{t_*+c_{min}}((m, \beta)\text{-LIS}, \mathcal{A}) - y\ .$$

Consequently,

$$\begin{aligned}
\mathcal{T}_{t^*}((m, \beta)\text{-LIS}, \mathcal{A}) &\leq \mathcal{T}_{t_*+c_{min}}((m, \beta)\text{-LIS}, \mathcal{A}) - y \\
&\leq \big(\mathcal{T}_{t_*+c_{min}}(\text{OPT}, \mathcal{A}) + \beta m^2 + 2m\big) - y \\
&\leq \mathcal{T}_{t^*}(\text{OPT}, \mathcal{A}) + (\beta m^2 + 2m) + m \\
&\leq \mathcal{T}_{t^*}(\text{OPT}, \mathcal{A}) + \beta m^2 + 3m
\end{aligned}$$

as desired. This contradicts the initial definition of time t^* in the proof sketch, and hence $\mathcal{T}_{t^*}((m, \beta)\text{-LIS}) \leq \mathcal{T}_{t^*}(\text{OPT}) + \beta m^2 + 3m$, from which the competitiveness for the number of pending tasks follows directly. As for the result of pending cost competitiveness, it is a direct consequence of the one for pending tasks, as the cost of any pending task in (m, β)-LIS is at most $\frac{c_{max}}{c_{min}} = \rho$ times bigger than the cost of any pending task in OPT. □

Observe that algorithm (m, β)-LIS uses the parameter β explicitly, which is critical for the proof of Lemma 7. According to its value, and if there are enough tasks in the queue, it achieves complete redundancy avoidance. More precisely, redundancy is avoided when β is no smaller than ρ and there are more than βm^2 tasks pending. If (some upper bound on) ρ is not available to the algorithm, then an inaccurate estimate of the value of β might not lead to complete redundancy avoidance, causing the above claimed competitiveness (and its proof) not to hold. We conjecture that even without knowing ρ it is still possible to obtain good competitiveness; this investigation is the subject of future work.

Algorithm 2 γ**m-Burst** (for machine p)

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

6. Algorithm γm-Burst

Consider an adversarial strategy that at the beginning of the execution injects only one c_{max}-task and then continues only with c_{min}-task injections. When algorithm (m, β)-LIS runs in a system with one machine under such an adversary and condition C1 holds (speedup $s < \rho$), it will have unbounded competitiveness. This is true due to the algorithm's nature to insist on scheduling the same task over and over again when stopped by a crash. An optimal algorithm on the other hand, would execute the task with the appropriate size in each alive interval of the machine. What is more, this can be generalized for m machines, and it is also the case for algorithms that use other scheduling policies, e.g. scheduling first the more costly tasks. This suggests that when condition C1 holds, a scheduling policy that alternates executions of lower cost tasks and higher cost ones should be devised.

In this section, we show that if the speedup satisfies condition C1 $\wedge \neg$C2, which implies $1 + \gamma/\rho \leq s < \rho$, and the tasks can have only two different costs, c_{min} and c_{max}, then there is an algorithm, call it γm-Burst, that achieves 1-pending-task and 1-pending-cost competitiveness in a system with m machines. See the algorithm's pseudocode (Algorithm 2) for details.

We first overview the main idea behind the algorithm. Each machine groups the set of pending tasks into two sublists, L_{min} and L_{max}, each corresponding to the tasks of cost c_{min} and c_{max} respectively, ordered by their arrival time. Following the same idea behind Algorithm (m, β)-LIS, γm-Burst avoids redundancy when "enough" tasks are pending. Furthermore, the algorithm needs to take into consideration parameter γ and the bounds on speedup s. In particular, in the case that there exist enough c_{min}- and c_{max}-tasks (more than m^2 to be exact) each machine completes no more than γ consecutive c_{min}-tasks and then a c_{max}-task. This is equal to the time it takes for the same machine to complete 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 completed. Special care needs to be taken for all other cases, e.g., when there are more than m^2 tasks of cost c_{max} pending but less than m^2 tasks of cost c_{min}, etc.
Overview of the proof: We first define a class of scheduling algorithms to which γm-Burst belongs (namely GroupLIS(1)), and show that as long as there are enough pending tasks, the algorithms in this class do not execute the same task twice, avoiding redundant executions. Observe that γm-Burst attempts to alternate the execution of γ c_{min}-tasks with one c_{max}-task, and $s \geq 1 + \gamma/\rho = \frac{\gamma c_{min} + c_{max}}{c_{max}}$. Then, if there are enough pending c_{max}-tasks, γm-Burst completes at least roughly the same number as the optimal algorithms OPT. Similarly, if there are enough pending c_{min}-tasks, γm-Burst completes at least roughly the same number as the optimal algorithms OPT. Combining these results we derive the task and cost competitiveness bounds.

Definition 1. *We define the* ***absolute task execution*** *of a task τ to be the interval $[t, t']$ in which a machine p schedules τ at time t and reports its completion to the repository at t', without stopping its execution within the interval $[t, t')$.*

Definition 2. *We say that a scheduling algorithm is of type* ***GroupLIS***(β)*, $\beta \in \mathbb{N}$, if all the following hold:*

- *It classifies the pending tasks into classes where each class contains tasks of the same cost.*
- *It sorts the tasks in each class in increasing order with respect to their arrival time.*
- *If a class contains at least $\beta \cdot m^2$ pending tasks and a machine p schedules a task from that class, then it schedules the $(p \cdot \beta m)$th task in the class.*

The next lemmas state useful properties of algorithms of type GroupLIS.

Lemma 8. *For an algorithm A of type GroupLIS(β) and a time interval I in which a list L of tasks of cost c has at least $\beta \cdot m^2$ pending tasks, any two absolute task executions fully contained in I, of tasks $\tau_1, \tau_2 \in L$, by machines p_1 and p_2 respectively, must have $\tau_1 \neq \tau_2$.*

Proof. Suppose by contradiction, that two machine p_1 and p_2 schedule the same c-task, say $\tau \in L$, to be executed during the interval I. Let's assume times t_1 and t_2, where $t_1, t_2 \in I$ and $t_1 \leq t_2$, to be the times when each of the machines correspondingly, scheduled the task. Since any c-task takes time $\frac{c}{s}$ to be completed, then p_2 must schedule the task before time $t_1 + \frac{c}{s}$, or else it would contradict the property of the Dispatcher stating that each reported task is immediately removed from the set of pending tasks.
Since algorithm A is of type GroupLIS(β), we have that at time t_1, when p_1 schedules τ, the task's position on the list L is $p_1 \cdot \beta n$. In order for machine p_2 to schedule τ at time t_2, it must be at position $p_2 \cdot \beta m$. There are two cases we have to consider:
(1) If $p_1 < p_2$, then during the interval $[t_1, t_2]$, task τ must increase its position in the list L from $p_1 \cdot \beta m$ to $p_2 \cdot \beta m$, i.e., by at least βm positions. This can happen only in the case when new tasks are injected and are placed before τ. This, however, is not possible, since new c-tasks are appended at the end of the list. (Recall that in algorithms of type GroupLIS, the tasks in L are sorted in an increasing order with respect to their arrival times.)
(2) If $p_1 > p_2$, then during the interval $[t_1, t_2]$, task τ must decrease its position in the list by at least βm places. This may happen only in the case where at least βm tasks ordered before τ in L at time t_1, are completed and reported by time t_2. Since all tasks in list L are of the same cost c, and the considered interval has length $\frac{c}{s}$, each machine may complete at most one task during that time. Hence, at most $n - 1$ tasks of cost c may be completed, which are not enough to change τ's position from $p_1 \cdot \beta m$ to $p_2 \cdot \beta m$ (even when $\beta = 1$) by time t_2.
The two cases above contradict the initial assumption and hence the claim of the lemma follows. □

Lemma 9. *Let S be a set of tasks reported as completed by an algorithm A of type GroupLIS(β) in a time interval I, where $|S| > m$. Then at least $|S| - m$ such tasks have their absolute task execution fully contained in I.*

Proof. A task τ which is reported in I by machine p and its absolute task execution $\alpha \not\subseteq I$, has $\alpha = [t, t']$ where $t \notin I$ and $t' \in I$. Since p does not stop executing τ in $[t, t')$, only one such task may occur for p. Then, there can not be more than m such reports overall and the lemma follows. □

Consider the following two interval types, used in the remainder of the section. $\mathcal{T}_t^{\max}(A, \mathcal{A})$ and $\mathcal{T}_t^{\min}(A, \mathcal{A})$ denote the number of pending tasks of costs c_{max} and c_{min} respectively at time t, with algorithm A and under adversarial pattern $\mathcal{A}$. Consider two types of intervals:

I^+: any interval such that $\mathcal{T}_t^{\max}(\gamma\text{m-Burst}, \mathcal{A}) \geq m^2, \forall t \in I^+$

I^-: any interval such that $\mathcal{T}_t^{\min}(\gamma\text{m-Burst}, \mathcal{A}) \geq m^2, \forall t \in I^-$

Then, the next two lemmas follow from Lemma 8 and the fact that algorithm γm-Burst is of type GroupLIS(1).

Lemma 10. *All absolute task executions of c_{max}-tasks in Algorithm γm-Burst within any interval I^+ appear exactly once.*

Lemma 11. *All absolute task executions of c_{min}-tasks in Algorithm γm-Burst within any interval I^- appear exactly once.*

The above leads to the following upper bound on the difference in the number of pending c_{max}-tasks.

Lemma 12. *The number of pending c_{max}-tasks in any execution of γm-Burst, under any adversarial pattern $\mathcal{A}$, run with speedup $s \geq 1 + \gamma/\rho$, is never larger than the number of pending c_{max}-tasks in the execution of OPT plus $m^2 + 2m$.*

Proof. Fix an adversarial pattern $\mathcal{A}$ and consider, for contradiction, interval $I^+ = (t_*, t^*]$ as it was defined above, t^* being the first time when $\mathcal{T}_{t^*}^{\max}(\gamma\text{m-Burst}, \mathcal{A}) > \mathcal{T}_{t^*}^{\max}(\text{OPT}, \mathcal{A}) + m^2 + 2m$, and t_* being the largest time before t^* such that $\mathcal{T}_{t_*}^{\max}(\gamma\text{m-Burst}, \mathcal{A}) < m^2$.

We claim that the number of absolute task executions of c_{max}-tasks $\alpha \subset I^+$, by OPT, is no bigger than the number of c_{max}-task reports by γm-Burst in interval I^+. Since $s \geq 1+\gamma/\rho = \frac{\gamma c_{min}+c_{max}}{c_{max}}$, while machine p in OPT is running a c_{max}-task, the same machine in γm-Burst has time to execute $\gamma c_{min} + c_{max}$ tasks. But, by definition, within the interval I^+ there are at least m^2 c_{max}-task pending at all times, which implies the execution of Case 3 or Case 4 of the γm-Burst algorithm. This means that no machine may run $\gamma + 1$ consecutive c_{min}-tasks, as a c_{max}-task is guaranteed to be executed by one of the cases. Hence, as claimed, the number of absolute task executions of c_{max}-tasks by OPT in the interval I^+ is no bigger than the number of c_{max}-task reports by γm-Burst in the same interval.

Now let κ be the number of c_{max}-tasks reported by OPT. From Lemma 9, at least $\kappa - m$ such tasks have absolute task executions in interval I^+. From the above claim, for every absolute task execution of c_{max}-tasks in the interval I^+ by OPT, there is at least a completion of a c_{max}-task by γm-Burst which gives a 1-1 correspondence, so γm-Burst has at least $\kappa - m$ reported c_{max}-tasks in I^+. Also, from Lemma 9, we may conclude that there are at least $\kappa - 2m$ absolute task executions of c_{max}-tasks in the interval. Then from Lemma 8, γm-Burst reports at least $\kappa - 2m$ different tasks, while OPT reports at most κ.

Now let S_{I^+} be the set of c_{max}-tasks injected during the interval I^+, under adversarial pattern $\mathcal{A}$. Then, for the number of c_{max}-tasks pending at time t^*, it holds that $\mathcal{T}^{\max}|_{t^*}(\gamma\text{m-Burst}, \mathcal{A}) < m^2 + |S_{I^+}| - (\kappa - 2m)$, and since $\mathcal{T}_{t^*}^{\max}(\text{OPT}, \mathcal{A}) \geq |S_{I^+}| - \kappa$ we have a contradiction, which completes the proof. ☐

Theorem 4. *$\mathcal{T}_t(\gamma$m-Burst$, \mathcal{A}) \leq \mathcal{T}_t(OPT, \mathcal{A}) + 2m^2 + (3 + \lceil \rho/s \rceil)m$, for any time t and adversarial pattern $\mathcal{A}$.*

Proof. Consider any adversarial pattern $\mathcal{A}$ and for contradiction, the interval $I^- = (t_*, t^*]$ as defined above, where t^* is the first time when $\mathcal{T}_{t^*}(\gamma\text{m-Burst}, \mathcal{A}) > \mathcal{T}_{t^*}(\text{OPT}, \mathcal{A}) + 2m^2 + (3 + \lceil \rho/s \rceil)m$ and t_* being the largest time before t^* such that $\mathcal{T}^{\max}|_{t_*}(\gamma\text{m-Burst}, \mathcal{A}) < m^2$. Notice that t_* is well defined for Lemma 12, i.e., such time t_* exists and it is smaller than t^*.

We consider each machine individually and break the interval I^- into subintervals $[t, t']$ such that times t and t' are instances in which the counter c is reset to 0; this can be either due to a simple reset in the algorithm or due to a crash and restart of a machine. More concretely, the boundaries of such subintervals are as follows. An interval can start either when a reset of the counter occurs or when the machine (re)starts. On its side, an interval can finish due to either a reset of the counter or a machine crash. Hence, these subintervals can be grouped into two types, depending on how they end: Type (a) which includes the ones that end by a crash and Type (b) which includes the ones that end by a reset from the algorithm. Note that in all cases γm-Burst starts the subinterval scheduling a new task to the machine at time t, and that the machine is never idle in the interval. Hence, all tasks reported by γm-Burst as completed have their absolute task execution completely into the subinterval. Our goal is to show that the number of absolute task executions in each such subinterval with γm-Burst is no less than the number of reported tasks by OPT.

First, consider a subinterval $[t, t']$ of Type (b), that is, such that the counter c is set to 0 by the algorithm (in a line $c = 0$) at time t'. This may happen in algorithm γm-Burst in Cases 1, 3 or 4. However, observe that the counter cannot be reset in Cases 1 and 3 at time $t' \in I^-$ since, by definition, there are at least m^2 tasks of cost c_{min} pending during the whole interval I^-. Case 4 implies that there are also at least m^2 tasks of cost c_{max} pending in γm-Burst. This means that in the interval $[t, t']$ there have been κ c_{min} and one c_{max} absolute task executions, $\kappa \geq \gamma$. Then, the subinterval $[t, t']$ has length $\frac{c_{max}+\kappa c_{min}}{s}$, and OPT can report at most $\kappa + 1$ task completions during the subinterval. This latter property follows from $\frac{c_{max}+\kappa c_{min}}{s} = \frac{c_{max}+\gamma c_{min}}{s} + \frac{(\kappa-\gamma)c_{min}}{s} \leq (\gamma+1)c_{min} + (\kappa-\gamma)c_{min} \leq (\kappa+1)c_{min}$, where the first inequality follows from the definition of γ (see Section 4) and the fact that $s > 1$. Now consider a subinterval $[t, t']$ of Type (a) which means that at time t' there was a crash. This means that no c_{max}-task was completed in the subinterval, but we may assume the complete execution of κ tasks of cost c_{min} in γm-Burst. We show now that OPT cannot report more than κ task completions. In the case where $\kappa \geq \gamma$, then the length of the subinterval $[t, t']$ satisfies

$$t' - t < \frac{\kappa c_{min} + c_{max}}{s} \quad \leq \quad (\kappa+1)c_{min}.$$

In the case where $\kappa < \gamma$ then the length of the subinterval $[t, t']$ satisfies

$$t' - t < \frac{(\kappa+1)c_{min}}{s} \quad \leq \quad (\kappa+1)c_{min}.$$

Then in none of the two cases OPT can report more than κ tasks in subinterval $[t, t']$.

After splitting I^- into the above subintervals, the whole interval is of the form $(t_*, t_1][t_1, t_2]\ldots[t_l, t^*]$. All the intervals $[t_i, t_{i+1}]$ where $t = 1, 2, \ldots, l$, are included in the subinterval types already analysed. There are therefore two remaining subintervals to consider now. The analysis of subinterval $[t_l, t^*]$ is verbatim to that of an interval of Type (a). Hence, the number of absolute task executions in that subinterval with γm-Burst is no less than the number of reported tasks by OPT.

Let us now consider the subinterval $(t_*, t_1]$. Assume with γm-Burst there are κ absolute task executions fully contained in the subinterval. Also observe that at most one c_{max}-task can be reported in the subinterval (since then the counter is reset and the subinterval ends). Then, the length of the subinterval is bounded as

$$t_1 - t_* < \frac{(\kappa+1)c_{min} + c_{max}}{s}$$

(assuming the worst case that a c_{min}-task was just started at t_* and that the machine crashed at t_1 when a c_{max}-task was about to finish). The number of tasks that OPT can report in the subinterval is hence bounded by

$$\left\lceil \frac{(\kappa+1)c_{min} + c_{max}}{s c_{min}} \right\rceil = \left\lceil \frac{(\kappa+1) + \rho}{s} \right\rceil < \kappa + 1 + \left\lceil \frac{\rho}{s} \right\rceil .$$

This means that for every machine, the number of reported tasks by OPT might be at most the number of absolute task executions by γm-Burst fully contained in I^- plus $1 + \lceil \rho/s \rceil$. From this and Lemma 11, it follows that in interval I^- the difference in the number of pending tasks between γm-Burst and OPT has grown by at most $(1 + \lceil \rho/s \rceil)m$. Observe that at time t_* the difference between the number of pending tasks satisfied

$$\mathcal{T}_{t_*}(\gamma\text{m-Burst}, \mathcal{A}) - \mathcal{T}_{t_*}(\text{OPT}, \mathcal{A}) < 2m^2 + 2m,$$

This follows from Lemma 12, which bounds the difference in the number of c_{max}-tasks to $m^2 + 2m$, and the assumption that $\mathcal{T}^{\max}|_{t_*}(\gamma\text{m-Burst}, \mathcal{A}) < m^2$. Then, it follows that $\mathcal{T}_{t^*}(\gamma\text{m-Burst}, \mathcal{A}) - \mathcal{T}_{t_*}(\text{OPT}, \mathcal{A}) < 2m^2 + 2m + (1 + \lceil \rho/s \rceil)m = m^2 + (3 + \lceil \rho/s \rceil)m$, which is a contradiction. Hence, $\mathcal{T}_t(\gamma\text{m-Burst}, \mathcal{A}) \leq \mathcal{T}_t(\text{OPT}, \mathcal{A}) + 2m^2 + (3 + \lceil \rho/s \rceil)m$, for any time t and adversarial pattern $\mathcal{A}$, as claimed. □

The difference in the number of c_{max}-tasks between ALG and OPT can be bounded by $m^2 + 2m$ (see Lemma 12). This, and Theorem 4, yield the following bound on the pending cost of γm-Burst, which also implies that it is 1-pending-cost competitive.

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

Unlike Algorithm (m, β)-LIS, even though γm-Burst uses the two cost values (see Algorithm 2), there is a simple way for it to work without having that knowledge. Algorithm γm-Burst works only for the case that there are two different task costs, maintaining two lists for the tasks according to their cost value. Even if the values of c_{min} and c_{max} are not given, it can follow the following strategy and still be 1-pending-task competitive and 1-pending-cost competitive.

As long as the pending tasks are only of one cost, no specific scheduling policy is necessary and therefore the simplest strategy to be followed is the *Longest In System*. As soon as two tasks of different cost arrive in the system, the machines can distinguish them by looking at their specifications. The algorithm will extract the two values, c_{max} and c_{min}, calculate the value of γ and create the corresponding lists as seen in the pseudocode (Algorithm 2). Hence, the following observation.

Observation 1. *The analysis of Algorithm γm-Burst holds even in the case that the values of c_{min} and c_{max} are unknown to the algorithm.*

7. Conditions on Competitiveness and Non-competitiveness

As we have shown in the previous sections via algorithms (m, β)-LIS and γm-Burst, the condition $s \geq \min\{\rho, 1 + \gamma/\rho\}$ is sufficient for achieving competitiveness. Complementary, it is enough that this condition does not hold (Theorem 2) to have no competitiveness. Since this condition on s depends on γ, which implicitly depends on s, in this section we study in more detail the bounds for competitiveness and non-competitiveness that relate only s and ρ.

Upper bound on the speedup for non-competitiveness. Recall that the ratio $\rho = c_{max}/c_{min} \geq 1$. We now derive properties in ρ that guarantee the above condition. From the first part of the condition for non-competitiveness, Condition **C1**, it must hold that $s < \rho$. From the second part, Condition **C2**, we must have

$$s < 1 + \gamma/\rho = 1 + \left\lceil \frac{\rho - s}{s - 1} \right\rceil /\rho = 1 + \left(\left\lceil \frac{\rho - 1}{s - 1} \right\rceil - 1 \right) /\rho,$$

where the second equality follows from $\lceil \frac{\rho - s}{s-1} \rceil = \lceil \frac{\rho-1}{s-1} \rceil - 1$. This, leads to

$$s < \frac{\lceil \frac{\rho-1}{s-1} \rceil + \rho - 1}{\rho}. \tag{1}$$

Let s_1 be the smallest speedup that satisfies Eq. 1, then a lower bound on s_1 can be found by removing the ceiling, as

$$s_1 \geq \frac{\frac{\rho-1}{s_1-1} + \rho - 1}{\rho} \implies s_1 \geq 2 - 1/\rho.$$

Summarizing, if $s < \rho$, then the first part of the condition for non-competitiveness (Condition **C1**) holds, and if $s < 2 - 1/\rho$, then the second part of the condition for non-competitiveness (Condition **C2**) holds. It can be shown that $\rho \geq 2 - 1/\rho$ for $\rho \geq 1$. Hence, we have the following result.

Theorem 6. *Let $\rho \geq 1$. In order to have non-competitiveness, it is sufficient to set $s < 2 - 1/\rho$.*

Smallest speedup for competitiveness. As mentioned above, in order to have competitiveness, it is sufficient that conditions C1 and C2 do not hold simultaneously. This means that at least one of the conditions ($\neg$C1) $s \geq \rho$, or ($\neg$C2) $s \geq 1 + \gamma/\rho$, must hold, where $\gamma = \max\{\lceil \frac{\rho-s}{s-1} \rceil, 0\}$. To satisfy condition ($\neg$C1), the speedup s must satisfy $s \geq \rho = \frac{c_{max}}{c_{min}}$. Hence, the smallest value of s that guarantees that ($\neg$C1) holds is $s_1 = \rho$. In order to satisfy condition ($\neg$C2), when condition ($\neg$C1) is not satisfied (observe that when ($\neg$C1) holds, $\gamma = 0$), we have

$$s \quad \geq \quad \frac{\lceil \frac{\rho-1}{s-1} \rceil + \rho - 1}{\rho}. \tag{2}$$

Let s_2 be the smallest speedup that satisfies Eq. 2; then an upper bound can be obtained by adding one unit to the expression in the ceiling

$$s_2 < \frac{\frac{\rho-1}{s_2-1} + 1 + \rho - 1}{\rho} \implies s_2 < 1 + \sqrt{1 - 1/\rho}\,.$$

Let us denote $s_2^+ = 1 + \sqrt{1 - 1/\rho}$. Then, in order to guarantee competitiveness, it is enough to choose any $s \geq \min\{s_1, s_2\}$. Since there is no simple form of the expression for s_2, we can use s_2^+ instead, to be safe. Then:

Theorem 7. *Let $\rho \geq 1$. In order to have competitiveness, it is sufficient to set $s = s_1 = \rho$ if $\rho \in [1, \varphi]$, and $s = s_2^+ = 1 + \sqrt{1 - 1/\rho}$ if $\rho > \varphi$, where $\varphi = \frac{1+\sqrt{5}}{2}$ is the golden ratio.*

Proof. As mentioned before, a sufficient condition for competitiveness is $s \geq \min\{s_1, s_2^+\}$. Using calculus is it easy to verify that $s_1 = \rho \leq s_2^+$ if $\rho \leq \varphi$. □

8. Algorithm (m, β)-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 C2 from Theorem 2 for these two costs, and choose (m, β)-LIS with speedup ρ in case $\rho \leq s^*$ and γm-Burst with speedup s^* otherwise[4]. However, in the case of more than two different task costs we cannot use γm-Burst, and so far we could only rely on (m, β)-LIS with speedup ρ, which can be large.

In this section we design a "substitute" for algorithm γm-Burst, working for any finite set C of different task costs in the interval $[c_{min}, c_{max}]$, that is competitive for some fixed small speedup ($s \geq 7/2$ to be exact). Note that $s \geq 2$ is enough to guarantee that condition C2 does not hold. This algorithm can therefore be used when ρ is large.

We call the new algorithm *Largest_Amortized_Fit* (LAF for short). It is parametrized by n and $\beta \geq \rho$ and is more "geared" towards pending cost efficiency. In particular, each machine keeps the variable $total$, storing the total cost of tasks reported by machine p, since the last restart (recall that upon a restart machines have no recollection of the past). Each machine schedules a task from the list of pending tasks that have the largest cost that is not bigger than $total$ and is such that there are at least βm^2 tasks of that cost pending, for $\beta \geq \rho$. Then, it sorts the pending tasks of that cost using the Longest-in-System (LIS) policy before choosing the task to schedule. If there is no cost meeting these requirements, the machine schedules an arbitrary pending task. See the algorithm's pseudocode (Algorithm 3) for details.

This algorithm, together with algorithm (m, β)-LIS, guarantee competitiveness for speedup $s \geq \min\{\rho, 7/2\}$. In more detail, one could apply (m, β)-LIS with speedup ρ when $\rho \leq 7/2$ and (m, β)-LAF with speedup $7/2$ otherwise.

As we prove in the following theorem, in order for the algorithm to be competitive, the number of different costs of injected tasks, i.e. $|C|$, must be finite in the range $[c_{min}, c_{max}]$. Otherwise, the number of tasks of the same cost might never be larger than βm^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 machine (the continuous period between a restart and a crash of the machine) only a constant fraction of this period could be wasted (with respect to 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 8. *Algorithm (m, β)-LAF is 1-pending-cost competitive, and thus ρ-pending-task competitive, for speedup $s \geq 7/2$, provided the number of different costs of tasks in the execution is finite.*

Proof. Note that algorithm (m, β)-LAF is in the class of GroupLIS(β) algorithms, for $\beta \geq \rho$. Therefore Lemma 8 applies, and together with the algorithm specification it guarantees no redundancy in absolute task executions in case that one of the lists is at a size of at least βm^2.

[4]Note that s^* is upper bounded by 2, as explained in Section 7.

Algorithm 3 (m, β)**-LAF** (for machine p)

```
Parameters: $C, m, \beta$
$total \leftarrow 0$                                                    //Upon awaking or restart
Repeat
    Get from the Repository the set of pending tasks $Pending$;
    Create lists $L_x$ of $x-$tasks, $\forall x \in C$;
    If $\{x \in C : x \leq total$ and $|L_x| \geq \beta m^2\} \neq \emptyset$ then
        $x_{max} \leftarrow \arg\max\{x \in C : x \leq total$ and $|L_x| \geq \beta m^2\}$;
        Sort $L_{x_{max}}$ according to task arrival;
        execute task $p \cdot \beta m$ in $L_{x_{max}}$;
        $total \leftarrow total + x_{max}$;
    else
        execute a random task $w$ in $Pending$;
        $total \leftarrow total + w.cost$;
    Inform the Repository of the task executed.
```

Consider any adversarial pattern $\mathcal{A}$. We show now that

$$\mathcal{C}_t^*((m,\beta)\text{-LAF}, \mathcal{A})|_{\geq x} \leq \mathcal{C}_t^*(\text{OPT}, \mathcal{A})|_{\geq x} + 2c_{max}k\beta m^2 + 2mc_{max} + 3mc_{max}/s$$

for every cost x at any time t and for speedup s, where $\mathcal{C}_t^*(\text{ALG}, \mathcal{A})|_{\geq x}$ denotes the sum of costs of pending tasks of cost at least x, and such that the number of pending tasks of such cost is at least βm^2 in (m, β)-LAF at time t of the execution of algorithm ALG, under adversarial pattern $\mathcal{A}$; k is the number of the possible different task costs that is injected under adversarial pattern $\mathcal{A}$. Note that this implies the statement of the theorem, since if we take x equal to the smallest possible cost and add an upper bound $c_{max}k\beta m^2$ on the cost of tasks on pending lists of (m, β)-LAF of size smaller than βm^2, we obtain the upper bound on the amount of pending cost of (m, β)-LAF, for any adversarial pattern $\mathcal{A}$.

Let us fix some cost x, and an adversarial pattern $\mathcal{A}$. Assume now, by contradiction, that the sought property does not hold, and let t^* be the first time t when $\mathcal{C}_t^*((m,\beta)\text{-LAF}, \mathcal{A})|_{\geq x} > \mathcal{C}_t^*(\text{OPT}, \mathcal{A})|_{\geq x} + 2c_{max}k\beta m^2 + 2mc_{max} + 3mc_{max}/s$ for task cost x and under the adversarial pattern $\mathcal{A}$. Denote by t_* the largest time before t^* such that for every $t \in (t_*, t^*]$, the following holds:

$$\mathcal{C}_t^*((m,\beta)\text{-LAF}, \mathcal{A})|_{\geq x} \geq \mathcal{C}_t^*(\text{OPT}, \mathcal{A})|_{\geq x} + c_{max}k\beta m^2$$

. Observe that t_* is well-defined, and moreover, $t_* \leq t^* - (c_{max} + 3c_{max}/s)$: it follows from the definition of t^* and from the fact that within a time interval $(t, t^*]$ of length smaller than $c_{max} + 3c_{max}/s$, OPT can report tasks of total cost at most $2mc_{max} + 3nc_{max}/s$, plus additional cost of at most $c_{max}k\beta m^2$ that can be caused by other lists growing beyond the threshold βm^2, and thus starting to contribute to the cost $\mathcal{C}^*$.

Consider now, interval $(t_*, t^*]$. By the specification of t_*, at any time of the interval there is at least one list of pending tasks of cost at least x that has length at least βm^2. Consider a life period of a machine p that starts in the considered time interval; let us restrict our consideration of this life period only by time t^*, and let c be the length of this period. Let $z > 0$ be the total cost of tasks, when counted only those of cost at least x, reported by machine p in the execution of OPT in the considered life period. We argue that in the same time interval, the total cost of tasks, when counted only those of cost at least x, reported by p in the execution of (m, β)-LAF is at least z. Observe that once machine p in (m, β)-LAF schedules a task of cost at least x for the first time in the considered period, it continues scheduling a task of cost at least x until the end of the considered period. Therefore, with respect to the corresponding execution of OPT, machine p could only waste its time (from perspective of executing a task of cost smaller than x or executing a task not reported in the considered period) in the first less than $(2x)/s$ time of the period or the last less than $(c/2)/s$ time of the period. Therefore, in the remaining period of length bigger than $c - (c/2 + 2x)/s$, machine

p is able to complete and report tasks, each of cost at least x, of total cost larger than

$$sc - (c/2 + 2x) \geq c(s - 1/2 - 2) \geq c \geq z ;$$

here in the first inequality we used the fact that $c \geq x$, which follows from the definition of $z > 0$, and in the second inequality we used the property $s - 1/2 - 2 \geq 1$ for $s \geq 7/2$. Applying Lemma 7, justifying no redundancy in absolute tasks executions of (m, β)-LAF in the considered time interval, we conclude life periods as considered do not contribute to the growth of the difference between $\mathcal{C}^*((m, \beta)\text{-LAF}, \mathcal{A})|_{\geq x}$ and $\mathcal{C}^*(\text{OPT}, \mathcal{A})|_{\geq x}$.

Therefore, only life periods that start before t_* can contribute to the difference in costs. However, if their intersections with the time interval $(t_*, t^*]$ is of length c at least $(2x + c_{max})/s$, that is, enough for a machine running (m, β)-LAF to report at least one task of length at least x, the same argument as in the previous paragraph yields that the total cost of tasks of cost at least x reported by a machine in the execution of (m, β)-LAF is at least as large as in the execution of OPT, minus the cost of the very first task reported by each machine in (m, β)-LAF (which may not be an absolute task execution and thus there may be redundancy on them) — i.e., minus at most mc_{max} in total. In the remaining case, i.e., when the intersection of the life period with $(t_*, t^*]$ is smaller than $(2x + c_{max})/s$, the machine may not report any task of length x when running (m, β)-LAF, but when executing OPT the total cost of all reported tasks is smaller than $(2x + c_{max})/s \leq 3c_{max}/s$. Therefore, the difference in costs on tasks of cost at least x between OPT and (m, β)-LAF could grow by at most $mc_{max} + 3mc_{max}/s$ in the life periods considered in this paragraph. Hence, $\mathcal{C}^*_{t^*}((m, \beta)\text{-LAF}, \mathcal{A})|_{\geq x} - \mathcal{C}^*_{t^*}(\text{OPT}, \mathcal{A})|_{\geq x} \leq \mathcal{C}^*_{t_*}((m, \beta)\text{-LAF}, \mathcal{A})|_{\geq x} - \mathcal{C}^*_{t_*}(\text{OPT}, \mathcal{A})|_{\geq x} + mc_{max} + 3mc_{max}/s \leq c_{max}k\beta m^2 + mc_{max} + 3mc_{max}/s$, which violates the initial contradictory assumption. $\square$

Observe that since Algorithm (m, β)-LAF uses parameter β in a similar manner as Algorithm (m, β)-LIS, its claimed competitiveness depends on the knowledge of (an upper bound on) ρ; recall relevant discussion at the end of Section 5.

9. Conclusions

Our major contribution shown in this paper is that a speedup $s \geq \min\{\rho, 1 + \gamma/\rho\}$ is *necessary* and *sufficient* to achieve competitiveness. In fact, we have shown that in order to have competitiveness, it is sufficient to set $s = \rho$ if $\rho \in [1, \varphi]$, and $s = 1 + \sqrt{1 - 1/\rho}$ otherwise, where $\varphi = \frac{1+\sqrt{5}}{2}$ is the golden ratio. For this, we have proposed and analysed *online* algorithms, which show that this speedup bound is sufficient for competitiveness. It is worth to observe that all our algorithms are *work-conserving*, meaning that they do not allow any processor to idle when there are pending tasks and never break a cycle. However, the negative results we give, on non-competitiveness, hold also for the case of non-work-conserving algorithms.

As discussed at the end of Sections 5 and 8, the claimed competitiveness of algorithms (m, β)-LIS and (m, β)-LAF is based on the knowledge of the values of the smallest and largest task costs, c_{min} and c_{max} (i.e., knowledge of a bound on ρ). A non-trivial future line, worth of investigation, would be to study in more detail the effects of this lack of knowledge on the algorithms' competitiveness. Another research line that we believe is worth of further investigation, is to study systems where processors can use different speedups, or systems where the speedup of processors could vary over time. Also, accommodating dependent tasks in the considered setting is also a challenging problem. Finally, as a future direction we are considering the problem under restricted adversaries (e.g., adversaries with bounded failure rate) or considering stochastic failures. We believe this could lead to better competitiveness bounds.

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 IEEE 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: extended abstract. 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 *Proceeding 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. Anderson and H. Woll. Algorithms for the certified write-all problem. *SIAM Journal on Computing*, 26(5):1277–1283, 1997.

[8] B. Awerbuch, S. Kutten, and D. Peleg. Competitive distributed job scheduling (extended abstract). In *Proceedings of the 24th ACM Symposium on Theory of computing (STOC 1992)*, pages 571–580, 1992.

[9] 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.

[10] 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.

[11] B.S. Chlebus, R. De Prisco, and A.A. Shvartsman. Performing tasks on synchronous restartable message-passing processors. *Distributed Computing*, 14(1):49–64, 2001.

[12] G. Cordasco, G. Malewicz, and A.L. 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.

[13] J. Dias, E. Ogasawara, D. de Oliveira, E. Pacitti, and M. Mattoso. Improving many-task computing in scientific workflows using p2p techniques. In *Proceedings of the 3rd IEEE Workshop on Many-Task Computing on Grids and Supercomputers*, 2010.

[14] Y. Emek, M.M. Halldórsson, 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 SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC 2010)*, pages 440–449, 2010.

[15] Enabling Grids for E-sciencE (EGEE). `http://www.eu-egee.org`.

[16] 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, 2011.

[17] Ch. Georgiou, A. Russell, and A.A. Shvartsman. The complexity of synchronous iterative Do-All with crashes. *Distributed Computing*, 17:47–63, 2004.

[18] Ch. Georgiou and A.A. Shvartsman. *Do-All Computing in Distributed Systems: Cooperation in the Presence of Adversity*. Springer, 2008.

[19] Ch. Georgiou and A.A. Shvartsman. *Cooperative Task-Oriented Computing: Algorithms and Complexity*. Synthesis Lectures on Distributed Computing Theory, Morgan & Claypool Publishers, 2011.

[20] 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.

[21] K.S. Hong and J.Y.-T Leung. On-line scheduling of real-time tasks. *IEEE Transactions on Computers*, 41(10):1326–1331, 1992

[22] K. Jeffay, D.F. Stanat, and C.U. Martel. On non-preemptive scheduling of period and sporadic tasks. In *Real-Time Systems Symposium, 1991. Proceedings., Twelfth*, pages 129–139, 1991.

[23] B. Joan and E. Faith. Bounds for scheduling jobs on grid processors. In Andrej Brodnik, Alejandro Lpez-Ortiz, Venkatesh Raman, and Alfredo Viola, editors, *Space-Efficient Data Structures, Streams, and Algorithms*, volume 8066 of *Lecture Notes in Computer Science*, pages 12–26. Springer Berlin Heidelberg, 2013.

[24] P.C. Kanellakis and A.A. Shvartsman. *Fault-Tolerant Parallel Computation*. Kluwer Academic Publishers, Norwell, MA, USA, 1997.

[25] S. Kentros, A. Kiayias, N. C. Nicolaou, and A. A. Shvartsman. At-most-once semantics in asynchronous shared memory. In *Proceedings of the 23rd International Symposium on Distributed Computing (DISC 2009)*, pages 258–273, 2009.

[26] E. Korpela, D. Werthimer, D. Anderson, J. Cobb, and M. Leboisky. Seti@home-massively distributed computing for seti. *Computing in Science Engineering*, 3(1):78–83, 2001.

[27] P. Lalanda. Shared repository pattern. In *Proceedings of the 5th Pattern Languages of Programs Conference (PLoP 1998)*, 1998.

[28] C.A. Phillips, C. Stein, E. Torng, and J. Wein. Optimal time-critical scheduling via resource augmentation (extended abstract). In *Proceedings of the 29th ACM Symposium on Theory of Computing (STOC 1997)*, pages 140–149, 1997.

[29] M.L. Pinedo. *Scheduling: Theory, Algorithms, and Systems*. Springer, fourth edition, 2012.

[30] K. Schwan and H. Zhou. Dynamic scheduling of hard real-time tasks and real-time threads. *IEEE Transactions on Software Engineering*, 18(8):736–748, 1992.

[31] D. Sleator and R.E. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28(2):202–208, 1985.

[32] U. van Heesch, S. Mahdavi 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.

[33] A. Wierman, L.L.H. Andrew, and A. Tang. Power-aware speed scaling in processor sharing systems. In *Proceedings of the IEEE INFOCOM 2009*, pages 2007–2015, 2009.

[34] F. Yao, A. Demers, and A. 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.

[35] 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.