Knowledge is Power: Online Performance of Non-uniform Tasks in Fault-prone Environments

Elli Zavou
Institute IMDEA Networks, 28918 Madrid, Spain
Universidad Carlos III de Madrid, 28911 Madrid, Spain

Abstract

Consider a system in which tasks of different execution times arrive continuously and have to be executed by a set of processors that are prone to crashes and restarts. In this work we explore the impact of parallelism and faults on the competitiveness of such a system. If the system had complete knowledge of future events and unbounded computation capability, it could make the best possible decisions and achieve optimal performance. Unfortunately, we show that no parallel deterministic algorithm can be competitive against the optimal solution provided in the idealistic scenario, even with tasks of only two different execution times. On the positive side, we show that providing additional energy to the system, in the form of processor speed-scaling, it is possible to develop deterministic algorithms that compare in favour to the optimal solution with complete knowledge. We identify thresholds on the speedup under which such competitiveness cannot be achieved by any deterministic algorithm and above which there exist competitive algorithms with small competitive ratio.

1 Introduction

Motivation. In recent years we have witnessed a dramatic increase on the demand of processing computationally-intensive jobs. Uniprocessors are no longer capable of coping with the high computational demands of such jobs. As a result, multicore-based parallel machines such as the K-computer [32] and Internet-based supercomputing platforms such as SETI@home [25] and EGEE Grid [18] have become the 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 computing entities 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.

For this purpose, much research, spanning the areas of Parallel&Distributed computing and Scheduling, was dedicated over the last two decades (e.g., [8, 14, 16, 17, 19, 20, 22, 24, 27, 31]). Different works address the issue by attacking different challenges. For example, many works address the issue of dynamic task injections, but do not consider failures (e.g., [11, 23, 31]). Others consider scheduling on one machine (e.g., [3, 28, 30]); with the drawback that the power of parallelism is not exploited (provided that tasks are independent). Some consider failures, but assume that tasks are known a priori and their number is bounded (e.g., [7, 12, 20, 24]), where other works assume that tasks are uniform, that is, they have the same processing times (e.g., [12, 19]). Several works consider power-preserving issues, but do not consider, for example, failures (e.g., [9, 11, 31]).

Contributions. In this work we consider a computing system in which tasks of *different* execution times arrive *dynamically and continuously* and must be performed by a set of n processors that are prone to *crashes and restarts*. Due to the dynamicity involved, we view this task-performing problem as an online problem and we pursue competitive analysis [29, 2]. Efficiency is measured as the maximum *pending cost* over any point of the execution, where pending cost is the sum of execution times of the tasks that have been injected in the system but have not been performed yet. We also account for the maximum *number of pending tasks* over any point of the execution. We believe that both efficiency measures are natural for the problem under study. E.g., the first measure is useful for evaluating the remaining processing time required from the system at any given point of the computation, and the second for evaluating the number of tasks still pending to be performed, regardless of the processing time needed. Our goal is to explore the impact of parallelism, the different task execution times, and the faulty environment on the competitiveness of the online system we consider.

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 processor *speed scaling* [5, 4, 11] is applied in the form of a *speedup* above a certain threshold. A speedup s means that a processor can perform a task s times faster than the task's system specified execution time. Speed scaling, impacts the *power consumption* of the processor. As a matter of fact, the power consumed to run a processor at a speed x grows superlinearly with x, and it is typically assumed to have a form of $P = x^{\alpha}$, for $\alpha > 1$[31, 1]. Hence, a speedup s implies an additional factor of s^{α} in the power consumed.

Therefore, our investigation aims at developing competitive online algorithms that require the smallest possible speedup. As a result, one of the main challenges of our work is to identify the speedup thresholds, over which competitiveness is possible and under which it is not possible. In some sense, our work can be seen as investigating the trade-offs between knowledge and power: How much power (in the form of speedup) does a deterministic scheduling algorithm need in order to match the efficiency of (i.e., be competitive with) the offline solution that possesses complete knowledge?

In summary, our contributions are as follows:

- We formalize an online task performing problem that abstracts important aspects of today's multicore-based parallel systems and Internet-based computing platforms: dynamic and continuous task injection, tasks with different processing times, processing entities subject to failures, and concerns on power-consumption. To the best of our knowledge, this is the first work to consider the problem and model as defined in this paper.
- We show that an offline version of a similar task-performing problem is NP-hard, both for pending cost and pending task efficiency, even if there is no parallelism (one processor), and the information of all tasks and processor availability is known. Since the offline problem is NP-hard, we concentrate on the online version and study competitiveness.

- We show *necessary* conditions on the value of the speedup s to achieve competitiveness. As mentioned above, identifying appropriate thresholds was a non-trivial task. We first considered a rather natural threshold: if tasks of cost at least c_{min} and at most c_{max} are injected in the system, $c_{min} \leq c_{max}$, then set the threshold to $s = \frac{c_{max}}{c_{min}}$. This implies that in the time needed from the offline algorithm to perform a task of cost c_{min}, a given online algorithm could perform a task of cost c_{max}. As it turns out, this threshold is not enough. So, we ended up considering another parameterized threshold. Let γ be the smallest integer where $\frac{\gamma c_{min}+c_{max}}{s} \leq (\gamma+1)c_{min}$. This parameter γ expresses the number of consecutive tasks of cost c_{min} together with a task of cost c_{max} that a given online algorithm can perform, in the same time period that the offline algorithm performs the same number of tasks of cost c_{min}. We define then an additional threshold of $s = \frac{\gamma c_{min}+c_{max}}{c_{max}}$.

  Then we show that if both conditions $s < \frac{c_{max}}{c_{min}}$ and $s < \frac{\gamma c_{min}+c_{max}}{c_{max}}$ hold, *no* deterministic sequential or parallel algorithm is competitive when run with speedup s. Furthermore, we show that these conditions are somewhat *minimal*, since, as we mention below, we present algorithms that achieve competitiveness as soon as any of them fails to hold.
- We then show *sufficient* conditions on s that lead to competitive solutions. In particular we develop algorithm (n, β)-LIS, and show that it achieves a bound on the number of pending tasks of $\mathcal{T}(\text{OPT}) + \beta n^2 + 3n$ for speedup $s \geq \frac{c_{max}}{c_{min}}$, for parameter $\beta \geq \frac{c_{max}}{c_{min}}$ and for any given n, where $\mathcal{T}(\text{OPT})$ denotes the number of pending tasks of the best offline algorithm that is aware a priori of the task injections and processor failures patterns. In order to be more cost competitive, (n, β)-LIS can be used with a higher speedup. Hence, we show a pending cost of $\mathcal{C}(\text{OPT}) + c_{max}\beta n^2 + (2c_{max} + c_{min})n$ for speedup $s \geq \frac{c_{max}}{c_{min}} \cdot \lceil \frac{c_{max}}{c_{min}} \rceil$, for parameter $\beta \geq \frac{c_{max}}{c_{min}}$ and for any given n, where $\mathcal{C}(\text{OPT})$ is the pending cost of OPT. Algorithm (n, β)-LIS balances between two paradigms: Longest In System task first (LIS) and redundancy avoidance. More precisely, the algorithm at a processor tries to schedule the task with longest pending time that does not cause redundancy of work if the number of pending tasks is sufficiently large.
- It is not difficult to observe that algorithm (n, β)-LIS cannot be competitive when $s < c_{max}/c_{min}$. To this respect, we develop algorithm γn-Burst and we show that when tasks of two different costs (c_{min}, c_{max}) are injected, the algorithm is competitive for $\frac{\gamma c_{min}+c_{max}}{c_{max}} \leq s < \frac{c_{max}}{c_{min}}$. Namely, it holds that $\mathcal{T}(\gamma\text{n-Burst}) \leq \mathcal{T}(\text{OPT}) + 2n^2 + (3 + \left\lceil \frac{c_{max}}{s \cdot c_{min}} \right\rceil)n,$, and $\mathcal{C}(\gamma\text{n-Burst}) \leq \mathcal{C}(\text{OPT}) + c_{max}(n^2 + 2n) + c_{min}(n^2 + (1 + \left\lceil \frac{c_{max}}{s \cdot c_{min}} \right\rceil)n)$. This investigation fully closes the gap with respect to the necessary conditions mentioned above.
- Next we develop algorithm LAF that is more "geared" towards pending cost efficiency. It generally follows the LIS paradigm, but it is adjusted based on the cost of the pending tasks. We show that this algorithm is competitive for speedup $s \geq \frac{7}{2}$. Hence, unlike the above mentioned algorithms, its competitiveness is with respect to a speedup that is independent of the values of the most costly and less costly tasks, c_{max} and c_{min}, respectively.
- Finally, we extend some of these results to a model in which the costs provided for the tasks may not be accurate, i.e., the actual cost may differ from the one given by a factor between $1 - \varepsilon$ and

$1+\varepsilon$, for some $\varepsilon \in [0,1)$. We show that many of the obtained results can be directly translated to this new model by simply replacing everywhere c_{min} by $(1-\varepsilon)c_{min}$ and c_{max} by $(1-\varepsilon)c_{max}$.

In order to keep our results conceptually general, we do not consider specific implementation details on how processors are informed about the tasks injected in the system. Instead, we consider an entity, called *dispatcher*, modeled as a shared object that processors can access by running specific operations. The dispatcher (whose detail specification is given in Section 2) abstracts the interface of the system in which clients submit computational tasks and notifies them when their tasks are completed. We note that the dispatcher is *not* a scheduler, that is, it does not make any task allocation decisions; it basically serves as a blackboard where processors can be informed about the set of pending tasks and inform back when they have completed some tasks. Hence, *the dispatcher* can be viewed as a much weaker abstraction of the master processor considered in Master-Worker Internet-based volunteering computing applications such as SETI@home [25], or the resource scheduler in computational Grid infrastructures such as EGEE [18], or the process scheduler in multicore-based platforms, e.g., [6], or the decision maker considered in the parallel scheduling literature, e.g., [27].

Related Work. The work most closely related to the present work is the one by Georgiou and Kowalski [19]. Similarly to this work, they consider a task-performing problem where tasks are dynamically and continuously injected to the system, and processors are subject to crashes and restarts. Unlike this work though, tasks are assumed to have *unit cost*, that is, they can be performed in one round. Furthermore, the computation is broken into synchronous rounds and the notion of *per-round* pending task competitiveness is considered instead. The authors first consider a version of the dispatcher (called central scheduler) and then show how and under what conditions it can be implemented in a message-passing distributed setting. They show that even in the presence of the central scheduler, no algorithm can be competitive if tasks have different execution times. This result has been the essential motivation of the present work; to use speed-scaling and study the conditions on speedup for which competitiveness is possible. As it turns out, extending the problem for tasks with different processing times and considering speed-scaling is a non-trivial task; different scheduling policies and techniques were needed to be devised.

The notion of competitiveness was introduced by Sleator and Tarjan [29] and was extended for parallel and distributed algorithms in a sequence of papers by Bartal et al. [10], Awerbuch et al. [8], and Ajtai et al. [2]. Several parallel and distributed computing problems have been modeled as online problems, and their competitiveness has been studied. Examples include distributed data management (e.g., [10]), distributed job scheduling (e.g., [8]), distributed collect (e.g., [13]), and set-packing (e.g., [17]).

In a sequence of papers [14, 15, 26], a scheduling theory is being developed for scheduling computations having intertask dependencies for Internet-based computing. The objective of the schedules is to render tasks eligible for execution at the maximum possible rate and avoid gridlock (although there are available computing elements, there are no eligible tasks to be performed). The task dependencies are represented as directed acyclic graphs and this line of work mainly focuses on exploiting the properties of DAGs in order to develop schedules. Even though our work considers independent tasks, it focuses on the development of fault-tolerant task performing algorithms, and explores the limitations of online distributed collaboration. However, extending our work to consider task dependencies would be an interesting and challenging direction.

Our work is related with works performed for parallel online scheduling using identical machines [27]. Among them, several ones consider speed-scaling and speedup issues. Some of them, unlike our work, consider dynamic scaling (e.g., [4, 9, 11]). Usually in these works, preemption is allowed: an execution of a task may be suspended and later restarted from the point of suspension. In our work, the task must be performed from scratch. The authors of [21] investigate scheduling on m identical speed-scaled processors without migration (tasks are not allowed to move among processors). Among others, they prove that any z-competitive online algorithm for a single processor yields a zB_a-competitive online algorithm for multiple processors, where B_a is the number of partitions of a set of size a. What is more, unlike our work, the number of processors is not bounded. The work in [5] considers tasks with deadlines (i.e., real-time computing is considered), but no migration, whereas the work in [4] considers both. We note that unlike our work, none of these works considers processor failures.

Document organization. The rest of the paper is structured as follows. In Section 2 the details of the model considered are described. Section 3 shows the NP-hardness of the offline problem of scheduling in an optimal way non-uniform tasks minimizing the pending cost and the number of pending tasks. In Section 4 it is shown that no competitiveness can be achieved if the speedup s is below the defined thresholds. Then, the algorithms (n, β)-LIS, γn-Burst, and LAF are presented and analyzed in Sections 5, 6, and 7, respectively. The model without accuracy is explored in Section 8 and finally, Section 9 concludes the outcome of our work. There is also a small Section 10 with some acknowledgements to the people that helped me complete this work.

2 Model and Definitions

Computing setting. We consider a system of n homogeneous, fault-prone processors, with unique ids from the set $[n] = \{1, 2, \ldots, n\}$. We assume that the processors have access to a shared object, called *dispatcher*. The dispatcher represents the interface of the system in which clients submit computational tasks and receive the notifications about the performed ones.

The data type of the dispatcher is a set of task specifications (their format is described later), that supports three operations: *inject, get,* and *inform.* The *inject* operation is executed by a client of the system. It adds a task specification to the current set; we assume that this operation is controlled by an adversary, whose characteristics are described later. The other two operations are executed by the processors. A processor, by executing a *get* operation, obtains a snapshot of the object, that is, a set of computational tasks. In particular, the dispatcher provides the set of *pending tasks*, that is, the tasks that have been injected into the system, but the dispatcher is not aware whether they have been performed. To simplify the model we assume that the *get* operation never returns an empty set of pending tasks, so if there are no pending tasks when the operation is executed, it blocks until some new task is injected, and then it immediately returns the set of new tasks. A processor, upon computing a certain task, executes an *inform* operation, which notifies the dispatcher about the completed task. The dispatcher then removes this task specification from the set of pending tasks.

Processors run in real-time cycles, controlled by an algorithm, where each cycle consists of: a *get* operation, a computation of a task, and an *inform* operation. Between two consecutive cycles an algorithm can choose to have a processor idling for a period of pre-defined 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, as described above). The computation part of the cycle, when it involves executing a task, consumes the time needed for the specific task to be computed divided by the *speedup* $s \geq 1$. An algorithm may decide to break the current cycle of a processor at any moment, in which case the processor starts a new one; a crash failure breaks (forcefully) the cycle of a processor, and upon a restart a new cycle begins, as discussed later. It is understood that if a processor does not perform any task then the *inform* operation is not executed. It is assumed that each operation performed by a processor is associated with a point in time (with the exception of a *get* that blocks), and the outcome of the operation is instantaneous (i.e., at the same time point). We say that an algorithm is *work conserving* if it does not allow any processor to be idle if there are pending tasks and it never breaks a cycle.

Note that processors communicate only by operations on the dispatcher, that is, they do not directly communicate between themselves. Also, due to the concurrent nature of the assumed computing system (i.e., processors' cycles may overlap between themselves and with the clients' inject operations), we specify the following event ordering at the dispatcher at a time t: the dispatcher first processes the *inform* operations executed by clients, then the *inject* operations, and last the *get* operations of processors. This implies that the set of pending tasks returned by a *get* operation executed at time t includes, besides the older unperformed tasks, the task specifications injected at time t and excludes the tasks reported as performed at time t. Note also, that this ordering of concurrent operations is done only for the ease of presentation and reasoning; it does not affect the generality of the claimed (non)competitiveness results.

Tasks. Each task specification τ is a tuple $(id,\ arrival,\ cost,\ code)$, where $\tau.id$ is a positive integer that uniquely identifies the task in the system, $\tau.arrival$ corresponds to the time at which the task was injected to the system (according to the dispatcher's local clock), $\tau.cost$ is the cost of the task, measured as the time needed for the task to be performed when running without a speedup, and $\tau.code$ corresponds to the computation that needs to occur so that the task is considered completed (that is, the computational part of the task specification that is actually performed). Throughout the paper we will be referring to a task of cost c as a c-task.

We assume that tasks are *atomic* with respect to their completion: if a processor stops executing a task, either intentionally or due to a crash, before completing the entire task, then not only the task is (obviously) not considered completed, but also no partial information can be shared with the dispatcher, nor the processor may resume the execution of the task from the point it stopped (i.e., it has to perform it from scratch; preemption is not allowed). Also note that if a processor performs a certain task, but it crashes before executing the *inform* operation, then this task is not considered completed.

Tasks are also assumed to be similar, independent and idempotent. By *similarity* we mean that the task computations on any processor consume equal or comparable local resources. By *independence* we mean that the completion of any task does not affect any other task, and any task can be performed concurrently with any other task. By *idempotence* we mean that each task can be performed one or more times to produce the same final result. Several applications involving tasks with such properties are discussed in [20]. Finally, we assume that task specifications are of polynomial size in n.

Adversary. We assume an adaptive and omniscient adversary that can cause crashes, restarts and task injections. We define an *adversarial pattern* $\mathcal{A}$ as a collection of crash, restart and injection events caused by the adversary. A $crash(t,i)$ event specifies that processor i is crashed at time t. A $restart(t,i)$ event specifies that processor i is restarted at time t; it is understood that no $restart(t,i)$ event can take place if there is no preceding $crash(t',i)$ event such that $t' < t$. Finally an $inject(t,\tau)$ event specifies that task specification τ is injected into the system (at the dispatcher) at time t. Let c_{min} and c_{max} denote the smallest and largest, respectively, costs that injected tasks may have. Unless stated otherwise, the adversary may inject tasks of *any* cost in the range $[c_{min}, c_{max}]$, and processors are not aware of c_{min} and c_{max}, i.e., these values are not provided to the processors as an input or a part of the code.

We say that a processor i is *alive* in time interval $[t,t']$, if the processor is operational at time t and does not crash by time t'. We assume that a restarted (or awaken) processor has knowledge of only the algorithm being executed and parameter n — the number of the system processors. Therefore, upon a restart, a processor simply starts a new cycle.

Efficiency measures. We evaluate our algorithms using the *pending cost* complexity measure, defined as follows. Given a time point $t \geq 0$ of the execution of an algorithm Alg, we define the *pending cost at time* t, $\mathcal{C}_t(\text{Alg})$, to be the sum of the costs of the pending tasks at the dispatcher at time t. Furthermore, we denote *the number of pending tasks* at the dispatcher at time t by $\mathcal{T}_t(\text{Alg})$. Then the *pending cost complexity* of algorithm Alg is defined as the maximum $\mathcal{C}_t(\text{Alg})$ over all time points (supremum in case of infinite computations). The *pending task complexity* is defined analogously.

Recall that in this work we view the task performance problem as an online problem, and hence we pursue competitive analysis. To this respect, we say that the *competitive pending cost complexity* is $f(\text{OPT}, n)$, for some function f, if and only if for every adversarial pattern $\mathcal{A}$ and time t, the pending cost at time t of the execution of the algorithm against adversarial pattern $\mathcal{A}$ is at most $f(\mathcal{C}_t(\text{OPT}(\mathcal{A})), n)$, where $\mathcal{C}_t(\text{OPT}(\mathcal{A}))$ is the minimum (or infimum, in case of infinite computations) pending cost achieved by an *off-line algorithm*, knowing $\mathcal{A}$, at time t of its execution under the adversarial pattern $\mathcal{A}$. The *competitive pending task complexity* is defined analogously.

It is worth to note here that as mentioned in Section 1, we prove that the problem of computing $\mathcal{C}_t(\text{OPT}(\mathcal{A}))$ and $\mathcal{T}_t(\text{OPT}(\mathcal{A}))$ offline, even for $n = 1$, is NP-hard (the proof is given in Section 3). We say that an algorithm *Alg* is *x-pending-cost competitive* if $\mathcal{C}_t(Alg, \mathcal{A}) \leq x \cdot \mathcal{C}_t(\text{OPT}, \mathcal{A}) + \Delta$, for any t and under any adversarial pattern $\mathcal{A}$; Δ can be any expression. Similarly, we say that an algorithm *Alg* is *x-pending-task competitive* if $\mathcal{T}_t(Alg, \mathcal{A}) \leq x \cdot \mathcal{T}_t(\text{OPT}, \mathcal{A}) + \Delta$.

Because, as we show later on, in the classical competitive measurement described above no competitiveness can be achieved by deterministic algorithms, we consider a speed-scaling approach in which the execution of an algorithm is done under processors speedup s, but it is compared to the minimum (infimum) pending cost achieved by off-line solutions run without speedup; each task takes the amount of time equal to its cost to be performed by the off-line solution. We study threshold values of s guarantying competitiveness, and specific competitiveness formulas are achieved.

Global notation. Throughout the paper, we will be using some global notation. We refer to the cost of a task τ by c_τ. As previously defined, c_{min} and c_{max} are the smallest and largest costs that tasks might have, respectively. The speedup of processors is represented by s. To refer to a time point in an execution we use t and to denote a time interval I. The cost of the tasks injected in an interval I is $\mathcal{C}_I$ whereas the set of injected tasks in the interval is S_I. The number of pending tasks of an algorithm Alg at time t is represented by $\mathcal{T}_t(\text{Alg})$ whereas the set of pending tasks is $Pending_t(\text{Alg})$ and their total pending cost is $\mathcal{C}_t(Alg)$.

3 NP-hardness

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

We consider two problems, depending whether the parameter to be minimized is the pending cost or the number of pending tasks. We call these problems *C_SCHED* and *T_SCHED*, respectively. The problems have as input:

- A set S of tasks to be executed. The cost of each tasks is also given.
- A checkpoint time t_{cp}. This is the point in time in which the amount of pending cost or the number of pending tasks will be evaluated.
- A processor activation schedule σ, which gives (until time t_{cp}) the time instants at which the processor will crash or restart.

The algorithm that solves the *C_SCHED* (resp. *T_SCHED*) problem, schedules the tasks so that the pending cost (resp. number of pending tasks) at time t_{cp} is minimized.

To prove NP-hardness we consider a decision version of these problems, called *DEC_C_SCHED* and *DEC_T_SCHED*. These problems have an additional input parameter ω. An algorithm that solves *DEC_C_SCHED* (resp., *DEC_T_SCHED*) outputs a Boolean value so that it outputs *TRUE* if and only if there is a schedule that achieves that the pending cost (resp., number of pending tasks) at time t_{cp} is no more than the parameter ω.

Theorem 1 *The problems DEC_C_SCHED and DEC_T_SCHED are NP-hard.*

Proof: We use the same reduction to prove the NP-hardness of both problems. The reduction is from the Partition problem. The input of the Partition problem is a set of numbers (we assume are positive) $C = \{x_1, x_2, ..., x_k\}$, $k > 1$. The problem is to decide whether 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_c of *DEC_C_SCHED* and an instance I_t of *DEC_T_SCHED* as follows (both instances have the same input). The set S is a set of k tasks, so that the ith task has cost x_i. The checkpointing time is $t_{cp} = 1 + \sum_{x_i \in C} x_i$. The schedule σ starts the processor at time 0 and crashes it at time $\frac{1}{2}\sum_{x_i \in C} x_i$. Then, σ restarts it immediately and crashes it again at time $\sum_{x_i \in C} x_i$. The processor does not restart until time t_{cp}. The parameter ω is set to $\omega = 0$.

Assume there is an algorithm A that solves *DEC_C_SCHED* (resp., *DEC_T_SCHED*). We show that A can be used to solve the instance I_p of Partition by solving the instance I_c of *DEC_C_SCHED* (resp., instance I_t of *DEC_T_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 processor is active, it is doing useful work. In that case, the pending cost (and, respectively, the number of pending tasks) at time t_{cp} will be $0 = \omega$. If, on the other hand, such subset does not exist, some of the time the processor is active will be wasted, and the cost (and number of tasks) pending at time t_{cp} has to be larger than ω. ■

4 Conditions on Non-competitiveness

For given task costs c_{min}, c_{max} and speedup s, we define parameter γ as the smallest non-negative integer that satisfies the following property:

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

It is not hard to derive that $\gamma = \max\{\lceil \frac{c_{max} - s c_{min}}{(s-1)c_{min}} \rceil, 0\}$. By definition, the following property is also satisfied:

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

Intuitively, γ gives the smallest number of c_{min}-tasks that an algorithm Alg with speedup s can complete in addition to a c_{max}-task, such that no algorithm running without speedup can complete more tasks in the same time.

We now prove necessary conditions for speedup value to achieve competitiveness. The main result of this section is as follows.

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

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

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

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

In other words, if $s < \min\left\{\frac{c_{max}}{c_{min}}, \frac{\gamma c_{min} + c_{max}}{c_{max}}\right\}$ then there is no competitive deterministic algorithm; we now prove this result.

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

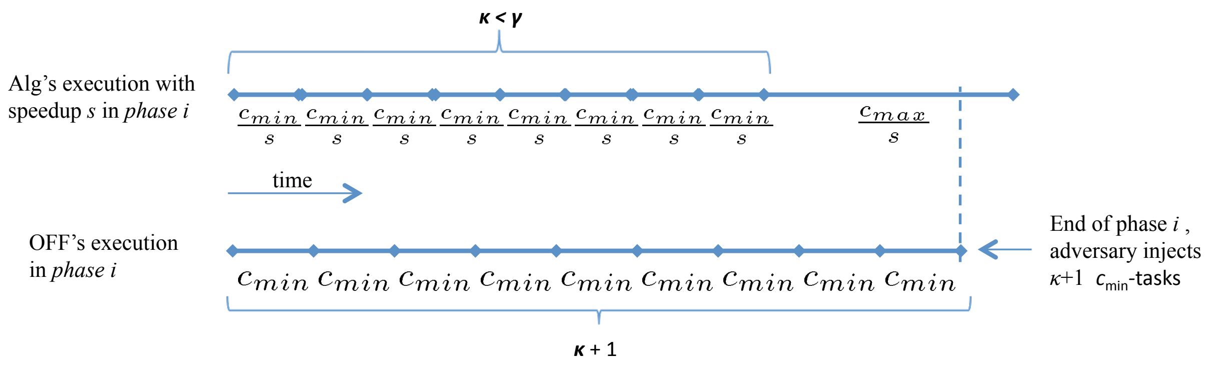

κ < γ
Alg's execution with speedup s in phase i
$\frac{c_{min}}{s}$ $\frac{c_{min}}{s}$ $\frac{c_{min}}{s}$ $\frac{c_{min}}{s}$ $\frac{c_{min}}{s}$ $\frac{c_{min}}{s}$ $\frac{c_{min}}{s}$ $\frac{c_{min}}{s}$
$\frac{c_{max}}{s}$
time
OFF's execution in phase i
c_{min} c_{min} c_{min} c_{min} c_{min} c_{min} c_{min} c_{min} c_{min}
End of phase i , adversary injects κ+1 c_{min}-tasks
κ + 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).

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

Suppose we have already defined adversarial pattern and algorithm OFF till the beginning of phase $i \geq 1$. Suppose that during the execution of Alg there are x of c_{min}-tasks and y of c_{max}-tasks pending. The adversary does not inject any tasks until the end of the phase. Under this assumption we could simulate the choices of Alg during the phase. There are two cases to consider (illustrated in Figures 1 and 2):

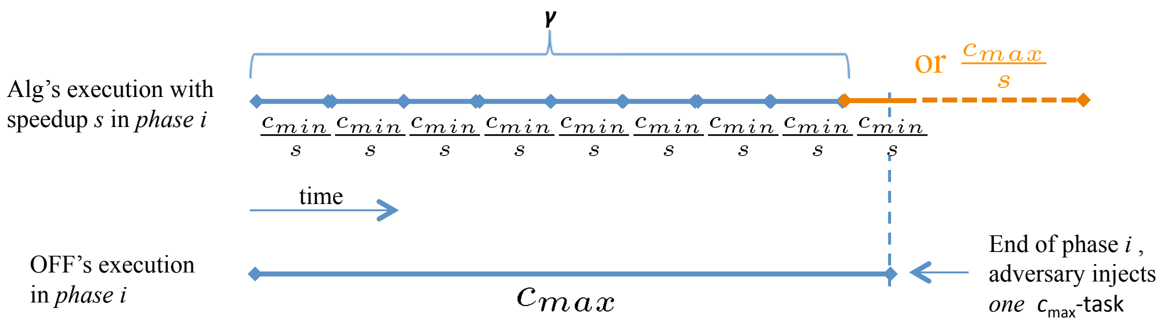
Scenario 1. Alg schedules κ of c_{min}-tasks, where $0 \leq \kappa < \gamma$, and then schedules a c_{max}-task; then OFF runs $\kappa + 1$ of c_{min}-tasks in the phase, and after that the processor is crashed and the phase is finished. At the end, $\kappa + 1$ of c_{min}-tasks are injected.

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

What remains to show is that the definitions of the OFF algorithm and the associated adversarial pattern are valid, and that in the execution of OFF the number of pending tasks is bounded while in the corresponding execution of Alg it is not bounded. But before we give a formal proof of these facts, let us state several useful properties of phases of the considered executions of Alg and OFF.

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

Proof: We argue by induction on the number of phases that: in 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



γ
or $\frac{c_{max}}{s}$
Alg's execution with speedup s in *phase i*
$\frac{c_{min}}{s}$ $\frac{c_{min}}{s}$ $\frac{c_{min}}{s}$ $\frac{c_{min}}{s}$ $\frac{c_{min}}{s}$ $\frac{c_{min}}{s}$ $\frac{c_{min}}{s}$ $\frac{c_{min}}{s}$ $\frac{c_{min}}{s}$
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 (b) of Theorem 2).

i is well defined due to the fact that its specification (including termination time) depends only if OFF can schedule either 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 tasks lengths that has been performed 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 are 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, as it is bounded by the time in which OFF performs either at most γ of c_{min}-tasks (in Scenario 1) or one c_{max}-task (in Scenario 2). Hence, the number of phases is infinite. ■

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

Proof: It follows from the specification of Scenarios 1 and 2, condition (b) on the speedup s, and from Property 2. Consider a phase. If Scenario 1 is applied for specification of this phase then Alg could not finish its c_{max}-task scheduled after $\kappa < \gamma$ c_{min}-tasks, because the time needed for competing this sequence of tasks is at least $\frac{\kappa c_{min}+c_{max}}{s}$, which, by Property 2, is larger than the length of this phase $(\kappa+1)c_{min}$. If Scenario 2 is applied for specification of this phase, 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 this phase c_{max}, by the assumption (b) on the speedup $s < \frac{\gamma c_{min}+c_{max}}{c_{max}}$. ■

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 phase i in the execution of Alg increases by one comparing with the beginning of phase i, while the number of pending c_{min}-tasks stay the same.*

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

Now we resume the main proof of non competitiveness, i.e., Theorem 2. By Lemma 1, the adversarial pattern and the corresponding offline algorithm OFF are well-defined. By Lemma 2, the number of phases is infinite. There are two cases. (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, and by Lemma 3 it never decreases, hence it is unbounded. (2) Otherwise, if the number of phases for which Scenario 2 was applied in the definition is bounded, after the last Scenario 2 phase in the execution of Alg, there are only phases in which Scenario 1 is applied, and there are infinitely many of them. In each such phase, Alg performs only κ of c_{min}-tasks while $\kappa + 1$ tasks will be injected at the end of the phase, for some corresponding non-negative integer $\kappa < \gamma$ defined in the specification of Scenario 1 for this phase. Indeed, the length of the phase is $(\kappa+1)c_{min}$, while after performing κ of c_{min}-tasks Alg schedules a c_{max}-task and is crashed before finishing it, because $\frac{\kappa c_{min}+c_{max}}{s} > (\kappa+1)c_{min}$ (c.f., 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 above cases the number of pending tasks in the execution of Alg grows unboundedly, while the numbed of pending tasks in the corresponding execution of OFF (for the same adversarial pattern) is bounded, by Lemma 1.

5 Algorithm (n, β)-LIS

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

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

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

Theorem 3 *Algorithm (n, β)-LIS has pending-task competitiveness of $\mathcal{T}_t(OPT) + \beta n^2 + 3n$ and pending-cost competitiveness of $\frac{c_{max}}{c_{min}} \cdot \big(\mathcal{C}_t(OPT) + \beta n^2 + 3n\big)$, for speedup $s \geq \frac{c_{max}}{c_{min}}$, when $\beta \geq \frac{c_{max}}{c_{min}}$.*

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

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

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: $\mathcal{T}_t((n,\beta)\text{-}LIS) \leq \mathcal{T}_t(OPT) + \beta n^2 + 2n$.*

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

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

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 (n,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 (n,β)-LIS in the same period plus n (counting possible redundancy).*

Proof: For each processor in the execution of OPT 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 n such tasks reported by OPT.

It remains to show that the number of remaining tasks reported to the dispatcher by OPT is not bigger than those reported in the execution of (n,β)-LIS in the considered period I. It follows from the property that $s \geq \frac{c_{max}}{c_{min}}$. More precisely, it implies that during time period when a processor p performs a task τ in the execution of OPT, the same processor reports at least one task to the Dispatcher in the execution of (n,β)-LIS. This is because performing any task by a processor in the execution of OPT takes at least time c_{min}, while performing any task by (n,β)-LIS takes no more than $\frac{c_{max}}{s} \leq c_{min}$, and also because no active processor in the execution of (n,β)-LIS is ever idle due to non-emptiness of the pending task queue. Hence we can define a 1-1 function from the considered tasks performed by OPT (i.e., tasks which are started and reported in time interval I) to the family of different tasks reported by (n,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 Dispatcher by (n, β)-LIS.*

Proof: The proof is by contradiction. Suppose that task τ is reported twice in the considered time interval of the execution of (n, β)-LIS. Consider the first two such reports, by processors 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 $c_{max}/s \le c_{min}$; otherwise it would mean that the processor who reported as the second would have started performing this task not earlier than the previous report to the dispatcher, which contradicts the property of the dispatcher 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 $p_2\beta n$. 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 βn^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 at the end of the list, it will never happen that a task with rank $p_1\beta n$ would increase its rank in time, in particular, not to $p_2\beta n$. 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 $p_1\beta n$ to $p_2\beta n$, i.e., by at least βn positions. It may happen only if at least βn 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 processor may report at most $\frac{c_{max}/s}{c_{min}/s} \le \beta$ tasks (this is the part of analysis requiring $\beta \ge \frac{c_{max}}{c_{min}}$). Since processor p_2 can report at most $\beta - 1$ tasks different than τ, the total number of tasks different from τ reported to the Dispatcher is at most $\beta n - 1$, and hence it is not possible to reduce the rank of τ from $p_1\beta n$ to $p_2\beta n$ within the considered time interval. This contradicts the assumption that p_2 reports τ to the Dispatcher at time t_2. ■

Lemma 8 *We have $\mathcal{T}_{t^*}((n,\beta)\text{-LIS}) \le \mathcal{T}_{t^*}(OPT) + \beta n^2 + 3n$.*

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

$$\mathcal{T}_{t^*}(\text{OPT}) \ge \mathcal{T}_{t_*+c_{min}}(\text{OPT}) - (y + n) \ .$$

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

$$\mathcal{T}_{t^*}((n,\beta)\text{-LIS}) \le \mathcal{T}_{t_*+c_{min}}((n,\beta)\text{-LIS}) - y$$

Consequently,

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

as desired. ■

We are now ready to show the proof of Theorem 3.

Proof of Theorem 3: The competitiveness for the number of pending tasks follows directly from Lemma 8: it violates the contradictory assumptions made in the beginning of the analysis. The result for the pending cost is a direct consequence of the one for pending tasks, as the cost of any pending task in (n,β)-LIS is at most $\frac{c_{max}}{c_{min}}$ times bigger than the cost of any pending task in OPT. ■

Looking at Theorem 3 one wonders whether it would be possible for algorithm (n,β)-LIS to be 1-pending cost competitive, for $s \geq \frac{c_{max}}{c_{min}}$. As it turns out, under certain conditions it cannot.

Theorem 4 *Even if $s \geq c_{max}/c_{min}$, if $c_{max} < \frac{2c_{min}}{s}$, then Algorithm (n,β)-LIS cannot be k-pending-cost competitive for any $k < \frac{1}{2}(\frac{c_{max}}{c_{min}}+1)$, even in the case of a single processor.*

Observe that $\frac{1}{2}(\frac{c_{max}}{c_{min}}+1) > 1$. The proof of this theorem (in fact of a more general result) is given in Appendix A.1.

On the contrary, if we allow $s \geq \frac{c_{max}}{c_{min}} \cdot \lceil \frac{c_{max}}{c_{min}} \rceil$ then algorithm (n,β)-LIS becomes 1-pending-cost competitive as well.

Theorem 5 *If $s \geq \frac{c_{max}}{c_{min}} \cdot \lceil \frac{c_{max}}{c_{min}} \rceil$, then $\mathcal{C}_t((n,\beta)\text{-}LIS) \leq \mathcal{C}_t(OPT) + c_{max}\beta n^2 + (2c_{max}+c_{min})n$, for $\beta \geq \frac{c_{max}}{c_{min}}$.*

The analysis of the algorithm in this case (including the proof of Theorem 5 above) is given in Appendix A.2.

6 Algorithm γn-Burst

Observe that against an adversarial strategy where a c_{max}-task is injected first and then only c_{min}-tasks are injected, algorithm (n,β)-LIS with one processor has unbounded competitiveness when $s < \frac{c_{max}}{c_{min}}$ (this can be generalized for n processors). This is also the case for algorithms using other scheduling policies, for example ones that schedule first the more costly tasks. This suggests that for $s < \frac{c_{max}}{c_{min}}$ a scheduling policy that alternates executions of lower-cost and higher-cost tasks should be devised. We investigate this here.

Specifically, we show that if the speed-up satisfies $\frac{\gamma c_{min}+c_{max}}{c_{max}} \leq s < \frac{c_{max}}{c_{min}}$ and the tasks can have only two different costs, c_{min} and c_{max}, then there is an algorithm, call it γn-Burst, that achieves 1-pending-task and 1-pending cost competitiveness in a system with n processors. Recall that $\gamma = \max\{\lceil \frac{c_{max}-sc_{min}}{(s-1)c_{min}} \rceil, 0\}$. It is easy to verify that if $s < \frac{c_{max}}{c_{min}}$ then $\gamma = \lceil \frac{c_{max}-sc_{min}}{(s-1)c_{min}} \rceil > 0$. This implies that $s > 1$.

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

Algorithm γn-Burst (for processor p)

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

We begin the analysis of the algorithm γn-Burst with some necessary definitions.

Definition 1 *We define the **absolute task execution** of a task τ to be the interval $[t, t']$ in which a processor p schedules τ at time t and reports its completion to the Dispatcher 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 arrival time.*
- *If a class contains at least $\beta \cdot n^2$ pending tasks and a processor p schedules a task from that class, then it schedules the $(p \cdot \beta n)$th task in the class.*

Observe that algorithm γn-Burst is of type GroupLIS (1). The next lemmas state useful properties of algorithms of type GroupLIS.

Lemma 9 *For an algorithm A of type GroupLIS (β) and a time interval I where a class L of cost c has at least $\beta \cdot n^2$ pending tasks, any two absolute task executions fully contained in I, of tasks $\tau_1, \tau_2 \in L$, by processors p_1 and p_2 respectively, must have $\tau_1 \neq \tau_2$.*

Proof: Suppose by contradiction, that two processors 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 processors 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 processor p_2 to schedule τ at time t_2, it must be at position $p_2 \cdot \beta n$. 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 $p_1 \cdot \beta n$ to $p_2 \cdot \beta n$, i.e., by at least βn positions. This can happen only in the case where 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 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 βn places. This may happen only in the case where at least βn 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 processor may complete at most one task during that time. Hence, at most $n - 1$ c-tasks may be completed, which are not enough to change τ's position from $p_1 \cdot \beta n$ to $p_2 \cdot \beta n$ (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 10 *Let S be a set of tasks reported as completed by an algorithm A of type GroupLIS (β) in a time interval I. Then at least $|S| - n$ such tasks have their absolute task execution fully contained in I.*

Proof: A task τ which is reported in I by processor 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, overall there can not be more than n such reports and the lemma follows.■

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

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

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

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

Lemma 12 *All absolute task executions of c_{min}-tasks in Algorithm γn-Burst within interval I^- appear exactly once.*

The above two lemmas follow from Lemma 9 and the fact that algorithm γn-Burst is of type GroupLIS (1). They also lead to the following upper bound on the difference in the number of pending c_{max}-tasks.

Lemma 13 *The number of pending c_{max}-tasks in any execution of γn-Burst, run with speed-up $s \geq \frac{\gamma c_{min}+c_{max}}{c_{max}}$, is never larger than the number of pending c_{max}-tasks in the execution of OPT plus n^2+2n.*

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

Claim: 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 γn-Burst in interval I^+.

Since $s \geq \frac{\gamma c_{min}+c_{max}}{c_{max}}$, while processor p in OPT is running a c_{max}-task, the same processor in γn-Burst has time to execute $\gamma c_{min} + c_{max}$ tasks. But, by definition, within the interval I^+ there are at least n^2 c_{max}-task pending at all times, which implies the execution of Case 3 or Case 4 of the γn-Burst algorithm. This means that no processor may run $\gamma+1$ consecutive c_{min}-tasks, as a c_{max}-task is guaranteed to be executed by one of the cases. So, 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 γn-Burst in the same interval. This completes the proof of the claim.

Now let κ be the number of c_{max}-tasks reported by OPT. From Lemma 10, at least $\kappa - n$ 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 γn-Burst which gives a 1-1 correspondence, so γn-Burst has at least $\kappa - n$ reported c_{max}-tasks in I^+. Also, from Lemma 10, we may conclude that there are at least $\kappa - 2n$ absolute task executions of c_{max}-tasks in the interval. Then from Lemma 9, γn-Burst reports at least $\kappa - 2n$ different tasks, while OPT reports at most κ.

Now let S_{I^+} be the set of c_{max}-tasks injected during the interval I^+. Then $\mathcal{T}^{\max}|_{t^*}(\gamma\text{n-Burst}) < n^2 + |S_{I^+}| - (\kappa - 2n)$, and since $\mathcal{T}^{\max}_{t^*}(\text{OPT}) \geq |S_{I^+}| - \kappa$ we have a contradiction, which completes the proof. ∎

Lemma 13 is then used to prove the following bound for both c_{max}- and c_{min}-tasks. This bounds implies that γn-Burst is 1-pending-task competitive.

Theorem 6 $\mathcal{T}_t(\gamma\text{n-Burst}) \leq \mathcal{T}_t(OPT) + 2n^2 + (3 + \left\lceil \frac{c_{max}}{s \cdot c_{min}} \right\rceil)n$*, for any time t.*

Proof: Consider, for contradiction, the interval $I^- = (t_*, t^*]$ as defined above, t^* being the first time when $\mathcal{T}_{t^*}(\gamma\text{n-Burst}) > \mathcal{T}_{t^*}(\text{OPT}) + 2n^2 + (3 + \left\lceil \frac{c_{max}}{s \cdot c_{min}} \right\rceil)n$ and t_* being the largest time before t^* such that $\mathcal{T}^{\max}|_{t_*}(\gamma\text{n-Burst}) < n^2$. Notice that t_* is well defined for Lemma 13, i.e., such time t_* exists and it is smaller than t^*.

We consider each processor 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 processor. 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 processor (re)starts. On its side, an interval can finish due to either a reset of the counter or a processor 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 γn-Burst starts the subinterval scheduling a new task to the processor at time t, and that the processor is never idle in the interval. Hence, all tasks reported by γn-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 γn-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 γn-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 n^2 c_{min}-tasks pending during the whole interval I^-. Case 4 implies that there are also at least n^2 c_{max}-tasks pending in γn-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 c_{min}-tasks in γn-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 $(t_*, t_1][t_1, t_2] \ldots [t_m, t^*]$. All the intervals $[t_i, t_{i+1}]$ where $t = 1, 2, \ldots, m$, are included in the subinterval types already analysed. There are therefore two remaining subintervals to consider now. The analysis of subinterval $[t_m, t^*]$ is verbatim to that of an interval of Type (a). Hence, the number of absolute task executions in that subinterval with γn-Burst is no less than the number of reported tasks by OPT.

Let us now consider the subinterval $(t_*, t_1]$. Assume with γn-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 processor 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

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

This means that for every processor, the number of reported tasks by OPT might be at most the number of absolute task executions by γn-Burst fully contained in I^- plus $1 + \left\lceil \frac{c_{max}}{s \cdot c_{min}} \right\rceil$. From this and Lemma 12, it follows that in interval I^- the difference in the number of pending tasks between γn-Burst and OPT has grown by at most $(1 + \left\lceil \frac{c_{max}}{s \cdot c_{min}} \right\rceil)n$. Observe that at time t_* the difference between the number of pending tasks satisfied

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

This follows from Lemma 13, which bounds the difference in the number of c_{max}-tasks to $n^2 + 2n$, and the assumption that $\mathcal{T}^{\max}|_{t_*}(\gamma\text{n-Burst}) < n^2$. Then, it follows that $\mathcal{T}_{t^*}(\gamma\text{n-Burst}) - \mathcal{T}_{t_*}(\text{OPT}) < 2n^2 + 2n + (1 + \left\lceil \frac{c_{max}}{s \cdot c_{min}} \right\rceil)n = n^2 + (3 + \left\lceil \frac{c_{max}}{s \cdot c_{min}} \right\rceil)n$, which is a contradiction. Hence, $\mathcal{T}_t(\gamma\text{n-Burst}) \leq \mathcal{T}_t(\text{OPT}) + 2n^2 + (3 + \left\lceil \frac{c_{max}}{s \cdot c_{min}} \right\rceil)n, \forall t$, as claimed. ■

Then, the combination of Lemma 13 and Theorem 6 yields the following bound on the pending cost of γn-Burst, which also implies that it is 1-pending-cost competitive.

Theorem 7 $\mathcal{C}_t(\gamma\textit{n-Burst}) \leq \mathcal{C}_t(\textit{OPT}) + c_{max}(n^2 + 2n) + c_{min}(n^2 + (1 + \left\lceil \frac{c_{max}}{s \cdot c_{min}} \right\rceil)n)$.

7 Algorithm LAF

In the case of only two different kinds of cost, we can obtain a competitive solution for speedup that matches the lower bound from Theorem 2. More precisely, for given two different cost values c_{min} and c_{max}, we can compute the minimum speedup s^* satisfying condition (b) from Theorem 2 for these two costs, and choose (n, β)-LIS with speedup c_{max}/c_{min} in case $c_{max}/c_{min} \leq s^*$ and γn-Burst with speedup s^* otherwise. However, in the case of more than two different task costs we cannot use γn-Burst, and so far we could only rely on (n, β)-LIS with speedup c_{max}/c_{min}, which can be large. In this section we design a "substitute" for algorithm γn-Burst, working for any bounded number of different task costs, which is competitive for speedup $s \geq 7/2$, and thus together with algorithm (n, β)-LIS guarantee competitiveness for speedup $s \geq \min\{\frac{c_{max}}{c_{min}}, 7/2\}$. In more detail, one could apply (n, β)-LIS with speedup $\frac{c_{max}}{c_{min}}$ when $\frac{c_{max}}{c_{min}} \leq 7/2$ and the new algorithm with speedup 7/2 otherwise.

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

In order for the algorithm to be feasibly implementable, the possible range of tasks needs to be discrete and bounded. Specifically, we assume that tasks of up to k different costs may be injected in the range c_{min}, c_{max}. However, it is not necessary for the algorithm to know a priori the different k costs; in fact it is not even necessary to know k. In Section 8 we discuss a possible way of making the algorithm work for unbounded number of different task costs.

Note that algorithm LAF is in the class of GroupLIS(β) algorithms, for $\beta \geq \frac{c_{max}}{c_{min}}$. Therefore Lemma 9 applies, and together with the algorithm specification it guarantees no redundancy in absolute tasks executions in case of one of the lists is kept of size at least βn^2. Then we obtain the following result.

Theorem 8 *Algorithm LAF is 1-pending-cost competitive, and thus $\frac{c_{max}}{c_{min}}$-pending-task competitive, for speedup $s \geq 7/2$.*

Proof: We show that $\mathcal{C}^*_t(\text{LAF})|_{\geq x} \leq \mathcal{C}^*_t(\text{OPT})|_{\geq x} + 2c_{max}k\beta n^2 + 2nc_{max} + 3nc_{max}/s$ for every cost x at any time t for speedup s, where $\mathcal{C}^*_t(Alg)|_{\geq x}$ 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 βn^2 in LAF at time t of the execution of algorithm Alg; k is the number of the possible different task costs that can be injected. 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 $c_{max}k\beta n^2$ on the cost of tasks on pending lists of LAF of size smaller than βn^2, we obtain the upper bound on the amount of pending cost of LAF.

Assume, to the contrary, that the sought property does not hold, and let t^* will be the first time t when $\mathcal{C}^*_t(\text{LAF})|_{\geq x} > \mathcal{C}^*_t(\text{OPT})|_{\geq x} + 2c_{max}k\beta n^2 + 2nc_{max} + 3nc_{max}/s$ for some cost x. Denote by t_* the largest time before t^* such that for every $t \in (t_*, t^*]$, $\mathcal{C}^*_t(\text{LAF})|_{\geq x} \geq \mathcal{C}^*_t(\text{OPT})|_{\geq x} + c_{max}k\beta n^2$. 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 $2nc_{max} + 3nc_{max}/s$, plus additional cost of at most $c_{max}k\beta n^2$ that can be caused by other lists growing beyond the threshold βn^2, and thus starting to contribute to the cost $\mathcal{C}^*$.

Consider 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 βn^2. Consider a life period of a process p that starts in the considered time interval; let us restrict our consideration of this life period only by time t^*, and 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 processor 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 LAF is at least z. Observe that once process p in LAF schedules a task of cost at least x for the first time in the considered period, it continues scheduling task of cost at least x until the end of the considered period. Therefore, with respect to the corresponding execution of OPT, processor p could only waste its time (from perspective of performing a task of cost smaller than x or performing 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 $c - (c/2 + 2x)/s$, processor 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 LAF in the considered time interval, we conclude life periods as considered do not contribute to the growth of the difference between $\mathcal{C}^*(\text{LAF})|_{\geq x}$ and $\mathcal{C}^*(\text{OPT})|_{\geq x}$.

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 processor running 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 processor in the execution of LAF is at least as large as in the execution of OPT, minus the cost of the very first task reported by each processor in LAF (which may not be an absolute task execution and thus there may be redundancy on them) — i.e., minus at most nc_{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 processor may not report any task of length x when running 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 LAF could grow by at most $nc_{max} + 3nc_{max}/s$ in the life periods considered in this paragraph. Hence, $\mathcal{C}^*_{t^*}(\text{LAF})|_{\geq x} - \mathcal{C}^*_{t^*}(\text{OPT})|_{\geq x} \leq \mathcal{C}^*_{t_*}(\text{LAF})|_{\geq x} - \mathcal{C}^*_{t_*}(\text{OPT})|_{\geq x} + nc_{max} + 3nc_{max}/s \leq c_{max}k\beta n^2 + nc_{max} + 3nc_{max}/s$, which violates the initial contradictory assumption. ■

8 Inaccuracy on the Task Costs

All algorithms presented in this paper ((n, β)-LIS, γn-Burst, and LAF) assume that the system provides accurate information with respect to the processing time (cost) needed for each task. But in reality, it could be the case that the system may provide only an estimate of the cost of the task. Hence this estimate might not be entirely accurate. We comment on this issue here.

Let us assume that the real cost of each task differs from the cost provided in the task specification by a factor of at least $1 - \varepsilon$ and at most $1 + \varepsilon$, for some $\varepsilon \in [0, 1)$. We do not assume that the knowledge about the value of ε is provided to processors as an input or part of the algorithm code. Then it is not difficult to see that the results presented in the previous sections can be adapted in this new setting. Lets refer to the new setting as *ε-inaccurate*. Note that in the ε-inaccurate setting, adversarial patterns in which the task costs included in tasks specifications are in some range $[c_{min}, c_{max}]$, the real task costs are in the range $[c_{min}(1 - \varepsilon), c_{max}(1 + \varepsilon)]$.

First observe that if the speed-up s satisfies both of the following conditions:

(a') $s < \frac{c_{max}(1+\varepsilon)}{c_{min}(1-\varepsilon)}$, and

(b') $s < \frac{\gamma' c_{min}(1-\varepsilon)+c_{max}(1+\varepsilon)}{c_{max}(1+\varepsilon)}$,

where $\gamma' = \max\{\lceil \frac{c_{max}(1+\varepsilon)-sc_{min}(1-\varepsilon)}{(s-1)c_{min}(1-\varepsilon)} \rceil, 0\}$, then no deterministic algorithm is competitive in the ε-inaccurate setting, even with one processor, run with speed-up s against an adversary injecting tasks with (inaccurate) cost in range $[c_{min}, c_{max}]$. This is because we can substitute c_{min} by $c_{min}(1-\varepsilon)$ and c_{max} by $c_{max}(1+\varepsilon)$ in the analysis of non-competitiveness in Section 4, which means that for the sake of non-competitiveness the adversary injects tasks of cost $c_{min}(1-\varepsilon)$ instead of c_{min}, still claiming in the task specifications that their cost is c_{min}, and tasks of cost $c_{max}(1+\varepsilon)$ instead of c_{max} with the cost in task specifications set to c_{max}.

Unfortunately, Algorithm γn-Burst is not adaptable (unless ε is known), as it relies on the values of c_{max} and c_{min}. Concerning Algorithm (n, β)-LIS, observe that the algorithm does not use any information about task costs, and therefore, using speedup $s \geq \frac{c_{max}(1+\varepsilon)}{c_{min}(1-\varepsilon)}$, its pending task competitiveness stays the same and its pending cost competitiveness becomes $\frac{c_{max}(1+\varepsilon)}{c_{min}(1-\varepsilon)} \cdot \big(\mathcal{C}_t(\text{OPT}) + \beta n^2 + 3n\big)$. Analogous result can be obtained for the case where $s \geq \frac{c_{max}(1+\varepsilon)}{c_{min}(1-\varepsilon)} \cdot \lceil \frac{c_{max}(1+\varepsilon)}{c_{min}(1-\varepsilon)} \rceil$.

However, the inaccuracy approach can be used by algorithm LAF to make it work for any range of task costs in $[c_{min}, c_{max}]$, and not just for k different costs. We describe now how to modify LAF to achieve this. Since the speedup s used does not depend on any parameter of the system it does not need to be changed. However, it is necessary to choose k reference cost values c_1 to c_k. We can define $c_i = c_{min} + \frac{c_{max}-c_{min}}{k}(i - \frac{1}{2})$. The idea is to divide the interval $[c_{min}, c_{max}]$ into k subintervals of equal width (for simplicity) $\frac{c_{max}-c_{min}}{k}$, and choose each c_i to be the cost in the middle of the ith subinterval. Let I_i denote the ith subinterval. Then, ε is chosen so that for each i from 1 to k, the subinterval I_i is fully contained into $[(1-\varepsilon)c_i, (1+\varepsilon)c_i]$. This value of ε can be easily computed.

Let us now assume a version of LAF that uses the k cost values defined, and assigns every injected task τ with cost c_τ to the cost c_i that minimizes $|c_\tau - c_i|$. Observe that then $c_\tau \in I_i \subseteq [(1-\varepsilon)c_i, (1+\varepsilon)c_i]$. Then this modified version of LAF still has 1-pending-cost competitiveness for any task cost domain.

9 Conclusions

In this paper we have shown that using a speedup $s \geq \min\left\{\frac{c_{max}}{c_{min}}, \frac{\gamma c_{min}+c_{max}}{c_{max}}\right\}$ is necessary and sufficient for competitiveness. In Appendix B we provide more insight on the boundary values of s for competitiveness and non-competitiveness. The main future lines we plan to explore are generalizations in which the speedup used is not homogeneous, but different processors can use different speedups or the speedup used can vary over time.

10 Acknowledgements

It would have been difficult to write my master thesis without the help and support of certain people around me, to only some of whom it is possible to give particular mention here.

I would like to express my deepest appreciation to my supervisor, Dr. Antonio Fernandez Anta, since without his help, support and patience I could not have written my thesis, not to mention his advices and encouragement at all times. I am also very grateful to Dr. Chryssis Georgiou from the University of Cyprus and Dr. Dariusz Kawalski from the University of Liverpool, whose help has been invaluable throughout all the research period. Last, I would like to thank Dr. Pablo Serrano who agreed to be my Tutor, as well as Dr. Alberto Garcia, Dr. Jose Alberto Hernandez and Dr. Angel Sanchez for being in the Committee.

References

[1] Enhanced intel speedstep technology for the intel pentium m processor. Intel White Paper 301170-001, 2004.

[2] M. Ajtai, J. Aspnes, C. Dwork, and O. Waarts. A theory of competitive analysis for distributed algorithms. In *Proceedings of the 35th Symposium on Foundations of Computer Science (FOCS 1994)*, pages 401–411, 1994.

[3] Susanne Albers and Antonios 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] Susanne Albers, Antonios Antoniadis, and Gero Greiner. On multi-processor speed scaling with migration. In *Proceedings of the 23rd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA 2011)*, pages 279–288, 2011.

[5] S. Anand, Naveen Garg, and Nicole 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.

[6] James H Anderson and John M Calandrino. Parallel real-time task scheduling on multicore platforms. In *Processings of the 27th IEEE International RealTime Systems Symposium (RTSS 2006)*, pages 89–100, 2006.

[7] R.J. Anderson and H. Woll. Algorithms for the certified Write-All problem. *SIAM Journal of Computing*, 26(5):1277–1283, 1997.

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

[9] Nikhil Bansal, Ho-Leung Chan, and Kirk 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] Y. Bartal, A. Fiat, and Y. Rabani. Competitive algorithms for distributed data management. In *Proceedings of the 24th ACM Symposium on Theory of Computing (STOC 1992)*, 1992.

[11] Ho Leung Chan, Jeff Edmonds, and Kirk Pruhs. Speed scaling of processes with arbitrary speedup curves on a multiprocessor. In *Proceedings of the 21st ACM Symposium on Parallelism in Algorithms and Architectures (SPAA 2009)*, pages 1–10, 2009.

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

[13] B.S. Chlebus, D.R. Kowalski, and A.A. Shvartsman. Collective asynchronous reading with polylogarithmic worst-case overhead. In *Proceedings of the 36th ACM Symposium on Theory of Computing (STOC 2004)*, pages 321–330, 1992.

[14] G. Cordasco, G. Malewicz, and A. Rosenberg. Advances in IC-Scheduling theory: Scheduling expansive and reductive dags and scheduling dags via duality. *IEEE Transactions on Parallel and Distributed Systems*, 18(11):1607–1617, 2007.

[15] G. Cordasco, G. Malewicz, and A. Rosenberg. Extending IC-Scheduling via the sweep algorithm. *Journal of Parallel and Distributed Computing*, 70(3):201–211, 2010.

[16] J. Dias, E. Ogasawara, D. de Oliveira, E. Pacitti, and M. Mattoso. A lightweight execution framework for massive independent tasks. In *Proceedings of the 3rd IEEE Workshop on Many-Task Computing on Grids and Supercomputers*, 2010.

[17] Y. Emek, M. M. Halldorsson, Y. Mansour, B. Patt-Shamir, J. Radhakrishnan, and D. Rawitz. Online set packing and competitive scheduling of multi-part tasks. In *Proceedings of the 29th ACM Symposium on Principles of Distributed Computing (PODC 2010)*, page 2010, 440–449.

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

[19] Chryssis Georgiou and Dariusz R. Kowalski. Performing dynamically injected tasks on processes prone to crashes and restarts. In *Proceedings of the 25th International Symposium on Distributed Computing, (DISC 2011)*, pages 165–180. Springer, 2011.

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

[21] Gero Greiner, Tim Nonner, and Alexander 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.

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

[23] K. Jeffay, D.F. Stanat, and C.U. Martel. On non-preemptive scheduling of period and sporadic tasks. In *Proceedings of the 12th Real-Time Systems Symposium*, pages 129–139, 1991.

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

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

[26] G. Malewicz, A. L. Rosenberg, and M. Yurkewych. Toward a theory for scheduling dags in internet-based computing. *IEEE Transactions on Computers*, 55(6):757–768, 2006.

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

[28] Karsten Schwan and Hongyi Zhou. Dynamic scheduling of hard real-time tasks and real-time threads. *IEEE Trans. Software Eng.*, 18(8):736–748, 1992.

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

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

[31] F. Frances Yao, Alan J. Demers, and Scott Shenker. A scheduling model for reduced CPU energy. In *Proceedings of the 36th IEEE Symposium on Foundations of Computer Science (FOCS 1995)*, pages 374–382, 1995.

[32] M. Yokokawa, F. Shoji, A. Uno, M. Kurokawa, and T. Watanabe. The k computer: Japanese next-generation supercomputer development project. In *Proceedings of the 2011 International Symposium on Low Power Electronics and Design (ISLPED 2011)*, pages 371–372, 2011.

APPENDIX

A Omitted Details from Sec. 5

A.1 LIS is not 1-cost-competitive

LIS schedules first the tasks that have been the longest waiting in the list of pending tasks. Let us consider a general class of LIS algorithms, that allow to schedule any task from the K oldest tasks, where K may depend of the system parameters (n, c_{max}, c_{min}, s, etc.). We call this class K-LIS. We show the following negative result which implies Theorem 9.

Theorem 9 *Even if $s \geq c_{max}/c_{min}$, if $c_{max} < \frac{2c_{min}}{s}$, then no algorithm of class K-LIS can be k-cost-competitive for any $k < \frac{1}{2}(\frac{c_{max}}{c_{min}} + 1)$, even in one single processor.*

Proof: We show that there is an execution with one single processor in which the cost competitiveness of K-LIS is no smaller than $\frac{1}{2}(\frac{c_{max}}{c_{min}} + 1)$. In this execution we will compare the pending cost of K-LIS, running with speedup s, with the pending cost of an algorithm LPT (*Largest Processing Time*), that always schedules tasks of length c_{max} if possible, and runs with no speedup. The adversary behaves as follows. It starts the processor at time 0 and crashes it at time c_{max}, restarts it immediately and crashes it again at time $2c_{max}$, restarts it immediately and crashes it again at time $3c_{max}$, and so on. The processor is hence active infinite number of intervals of length c_{max}, which we call *active intervals*. The adversary injects at time 0 one task of costs c_{max} and K tasks of cost c_{min}. Then, every time the processor is crashed two new tasks are injected, one c_{min}-task and one c_{max}-task, in this order.

Observe that LPT completes one c_{max}-task in each interval the processor is active. This is easy to see, since there is always a pending c_{max}-task and each interval is just long enough for the task to be completed. K-LIS, on the other hand, in the first active interval has no choice but start scheduling a c_{min}-task. After completing it, it may schedule the only pending c_{max}-task or another c_{min}-task. In either case, that task is not going to completed, since $c_{max} < \frac{2c_{min}}{s}$. Hence, K-LIS could only complete a c_{min}-task in that interval while LPT completed a c_{max}-task. If K-LIS starts an active interval scheduling a c_{max}-task, then the task is completed, but no other task can be completed in the interval. Observe then that (1) K-LIS completes at most one task in each active interval, and hence, from the injection pattern, (2) K-LIS gets a new c_{max}-task in the set of K oldest tasks at most once every two active intervals.

We can compute now the pending tasks under each algorithm after completing i active intervals. The total set of injected tasks contains the $K + 1$ tasks injected initially and the two tasks injected at the end of each active interval. These add up to $i + 1$ tasks of cost c_{max} and $K + i$ tasks of cost c_{min}. Of these tasks, LPT has completed and reported i tasks of cost c_{max}. To compute the tasks completed by K-LIS we first assume i even, for simplicity, and assume that K-LIS prefer to schedule c_{max}-tasks if there are among the K oldest tasks. (This is in fact the best strategy for K-LIS.) With these assumptions we have that K-LIS has completed and reported $i/2$ tasks of cost c_{max} and $i/2$ tasks of cost c_{min}.

The ratio $\rho_i = \mathcal{C}_{ic_{max}}(K\text{-LIS})/\mathcal{C}_{ic_{max}}(\text{LPT})$ of pending costs after i active intervals is then

$$\begin{aligned}\rho_i &= \frac{(i+1)c_{max} + (K+i)c_{min} - \frac{ic_{max}}{2} - \frac{ic_{min}}{2}}{(i+1)c_{max} + (K+i)c_{min} - ic_{max}} \\ &= \frac{(\frac{i}{2}+1)c_{max} + (K+\frac{i}{2})c_{min}}{c_{max} + (K+i)c_{min}}.\end{aligned}$$

Using calculus, it is easy to prove that this ratio ρ_i tends to $\frac{1}{2}(\frac{c_{max}}{c_{min}}+1)$ when i tends to infinity. This proves that K-LIS cannot be k-cost-competitive for any $k < \frac{1}{2}(\frac{c_{max}}{c_{min}}+1)$. ■

A.2 Analysis of (n,β)-LIS for speedup $s \geq \frac{c_{max}}{c_{min}} \cdot \lceil \frac{c_{max}}{c_{min}} \rceil$

If we allow speedup $s \geq \frac{c_{max}}{c_{min}} \cdot \lceil \frac{c_{max}}{c_{min}} \rceil$, algorithm (n,β)-LIS is OPT $+ c_{max}\beta n^2 + (2c_{max}+c_{min})n$ competitive in terms of pending cost, for $\beta \geq \frac{c_{max}}{c_{min}}$. Suppose otherwise and let fix the speedup $s \geq \frac{c_{max}}{c_{min}} \cdot \lceil \frac{c_{max}}{c_{min}} \rceil$. Consider an execution in which the pending cost of (n,β)-LIS, $\mathcal{C}((n,\beta)\text{-LIS})$, is bigger than $\mathcal{C}(\text{OPT}) + c_{max}\beta n^2 + (2c_{max}+c_{min})n$, and fix the adversarial pattern associated with it together with the optimum solution OPT for it. Let t^* be a time in the execution when $\mathcal{C}_{t^*}((n,\beta)\text{-LIS}) > \mathcal{C}_{t^*}(\text{OPT}) + c_{max}\beta n^2 + (2c_{max}+c_{min})n$ in terms of pending cost. For any time interval I, let $\mathcal{C}_I$ be the total cost of tasks injected in the interval I. Let $t_* \leq t^*$ be the smallest time such that for any $t \in [t_*, t^*)$, $\mathcal{C}_t((n,\beta)\text{-LIS}) > \mathcal{C}_t(\text{OPT}) + c_{max}\beta n^2$. (Note again that the selection of minimum time satisfying some properties defined by the computation is possible due to the fact that the computation is split into discrete processor cycles.) Observe that $\mathcal{C}_{t_*}((n,\beta)\text{-LIS}) \leq \mathcal{C}_{t_*}(\text{OPT}) + c_{max}\beta n^2 + c_{max}n$ because at time t_* no more than n tasks could be reported to the dispatcher by OPT, each of cost at most c_{max}, while just before t_* the difference between pending costs of (n,β)-LIS and OPT was at most $c_{max}\beta n^2$.

Lemma 14 *We have $t_* < t^* - c_{min}$, and for every $t \in [t_*, t_* + c_{min}]$ the following holds: $\mathcal{C}_t((n,\beta)\text{-}LIS) \leq \mathcal{C}_t(OPT) + c_{max}\beta n^2 + (c_{max}+c_{min})n$.*

Proof: Indeed, in the interval $[t_*, t_* + c_{min}]$, OPT can notify a dispatcher about at most $(c_{max} + c_{min}) \cdot n$ total cost completed, as each of n processors may finish a task of cost at most c_{max} in the beginning of the considered time interval and at most c_{min} cost units during the interval. Consider any $t \in [t_*, t_* + c_{min}]$ and let I be fixed to $[t_*, t]$. We have $\mathcal{C}_t((n,\beta)\text{-LIS}) \leq \mathcal{C}_{t_*}((n,\beta)\text{-LIS}) + \mathcal{C}_I$ and $\mathcal{C}_t(\text{OPT}) \geq \mathcal{C}_{t_*}(\text{OPT}) + \mathcal{C}_I - (c_{max}+c_{min})n$. It follows that $\mathcal{C}_t((n,\beta)\text{-LIS}) \leq \mathcal{C}_{t_*}((n,\beta)\text{-LIS}) + \mathcal{C}_I \leq \big(\mathcal{C}_{t_*}(\text{OPT}) + c_{max}\beta n^2\big) + (\mathcal{C}_t(\text{OPT}) - \mathcal{C}_{t_*}(\text{OPT}) + (c_{max}+c_{min})n) \leq \mathcal{C}_t(\text{OPT}) + c_{max}\beta n^2 + (c_{max} + c_{min})n$. It also follows that any such t must be smaller than t^*. ■

Lemma 15 *Consider a time interval I during which the queue of pending tasks in (n,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 (counting possible redundancy) reported by (n,β)-LIS in the same period divided by $\lceil \frac{c_{max}}{c_{min}} \rceil$ plus n.*

Proof: For each processor in the execution of OPT 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 n such tasks reported by OPT.

It remains to show that the number of remaining tasks reported to the dispatcher by (n,β)-LIS is at least $\lceil\frac{c_{max}}{c_{min}}\rceil$ times the number of those reported in the execution of OPT in the considered period. It follows from the property of $s \geq \frac{c_{max}}{c_{min}} \cdot \lceil\frac{c_{max}}{c_{min}}\rceil$. More precisely, it implies that during time period when a processor p performs a task τ in the execution of OPT, the same processor reports at least $\lceil\frac{c_{max}}{c_{min}}\rceil$ tasks to the Dispatcher in the execution of (n,β)-LIS. This is because performing any task by a processor in the execution of OPT takes at least time c_{min}, while performing any task by (n,β)-LIS takes no more than $c_{max}/s \leq \frac{c_{min}}{\lceil c_{max}/c_{min}\rceil}$, and also because no active processor in the execution of (n,β)-LIS is ever idle due to non-emptiness of the pending task queue. Hence we may define a 1-1 function from the considered tasks performed by OPT (i.e., tasks which are started and reported in time interval I) to the family of disjoint task sets of size at least $\lceil\frac{c_{max}}{c_{min}}\rceil$ reported by (n,β)-LIS in the period I, which completes the proof. ∎

Lemma 16 *We have $\mathcal{C}_{t^*}((n,\beta)\text{-}LIS) \leq \mathcal{C}_{t^*}(OPT) + c_{max}\beta n^2 + (2c_{max}+c_{min})n$.*

Proof: By Lemma 14 we have that $\mathcal{C}_{t_*+c_{min}}((n,\beta)\text{-LIS}) \leq \mathcal{C}_{t_*+c_{min}}(\text{OPT}) + c_{max}\beta n^2 + (c_{max} + c_{min})n$. Let x be the total cost of tasks injected in $(t_*+c_{min}, t^*]$, and let y be the total number of tasks reported by (n,β)-LIS in $(t_*+c_{min}, t^*]$. By Lemma 15, OPT reports no more that $\lceil\frac{c_{max}}{c_{min}}\rceil^{-1}y + n$ tasks in $(t_*+c_{min}, t^*]$. Therefore, in terms of pending cost, $\mathcal{C}_{t^*}(\text{OPT}) \geq \mathcal{C}_{t_*+c_{min}}(\text{OPT}) + x - \left(\lceil\frac{c_{max}}{c_{min}}\rceil^{-1}y + n\right)c_{max} \geq \mathcal{C}_{t_*+c_{min}}(\text{OPT}) + x - (yc_{min} + nc_{max})$.

By Lemma 7, in the interval $[t_*+c_{min}, t^*]$, no redundant work is reported by (n,β)-LIS. Thus, $\mathcal{C}_{t^*}((n,\beta)\text{-LIS}) \leq \mathcal{C}_{t_*+c_{min}}((n,\beta)\text{-LIS}) + x - yc_{min}$. $\mathcal{C}_{t^*}((n,\beta)\text{-LIS}) \leq \mathcal{C}_{t_*+c_{min}}((n,\beta)\text{-LIS}) + x - yc_{min} \leq \left(\mathcal{C}_{t_*+c_{min}}(\text{OPT}) + c_{max}\beta n^2 + (c_{max}+c_{min})n\right) + x - yc_{min} \leq (\mathcal{C}_{t^*}(\text{OPT}) - x + yc_{min} + nc_{max}) + (c_{max}\beta n^2 + (c_{max}+c_{min})n) + x - yc_{min} \leq \mathcal{C}_{t^*}(\text{OPT}) + c_{max}\beta n^2 + (2c_{max}+c_{min})n$, as desired. ∎

We now prove Theorem 5.

Proof of Theorem 5: Lemma 16 leads to contradiction and the claimed result follows. ∎

B Conditions on Competitiveness and Non-competitiveness

Upper bound on the speedup for non-competitiveness As proven in Theorem 2, the condition $s < \min\left\{\frac{c_{max}}{c_{min}}, \frac{\gamma c_{min}+c_{max}}{c_{max}}\right\}$ is sufficient for non competitiveness. Let us define ratio $\rho = c_{max}/c_{min} \geq 1$. We will derive properties in ρ that guarantee the above condition. From the first part (condition (a) in Theorem 2), it must hold that $s < \frac{c_{max}}{c_{min}} = \rho$. From the second part (condition (b) in Theorem 2), we must have

$$\begin{aligned} s &< \frac{\gamma c_{min} + c_{max}}{c_{max}} \\ &= \frac{\lceil\frac{c_{max}-sc_{min}}{c_{min}(s-1)}\rceil c_{min} + c_{max}}{c_{max}} \\ &= \frac{\lceil\frac{c_{max}-c_{min}}{c_{min}(s-1)}\rceil c_{min} + c_{max} - c_{min}}{c_{max}} \\ &= \frac{\lceil\frac{\rho-1}{s-1}\rceil + \rho - 1}{\rho}\ , \end{aligned} \tag{1}$$

where the second equality follows from $\lceil \frac{c_{max}-sc_{min}}{c_{min}(s-1)} \rceil = \lceil \frac{c_{max}-c_{min}}{c_{min}(s-1)} \rceil - 1$. Let s_1 be the smallest ρ that satisfies Eq. 1, then a lower bound on s_1 can be found by removing the ceiling, as

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

It can be shown that $\rho \geq 2 - 1/\rho$ for $\rho \geq 1$. Then, a sufficient condition for non competitiveness is

$$s < 2 - 1/\rho = 2 - c_{min}/c_{max}.$$

Smallest speedup for competitiveness As we show in this work, in order to have competitiveness, $s \geq \min \left\{ \frac{c_{max}}{c_{min}}, \frac{\gamma c_{min}+c_{max}}{c_{max}} \right\}$ is sufficient. This means that (a) $s \geq \frac{c_{max}}{c_{min}}$, or (b) $s \geq \frac{\gamma c_{min}+c_{max}}{c_{max}}$ must hold, where $\gamma = \max\{\lceil \frac{c_{max}-sc_{min}}{(s-1)c_{min}} \rceil, 0\}$. To satisfy condition (a), the speedup s must satisfy $s \geq \frac{c_{max}}{c_{min}} = \rho$. Hence, the smallest value of s that guarantees that (a) holds is $s_{(a)} = \rho$.

In order to satisfy condition (b), when condition (a) is not satisfied (observe that when (a) holds, $\gamma = 0$), we have

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

Let $s_{(b)}$ be the smallest ρ that satisfies Eq. 2; then an upper bound can be obtained by adding one unit to the expression in the ceiling

$$s_{(b)} < \frac{\frac{\rho-1}{s_{(b)}-1} + 1 + \rho - 1}{\rho} \implies s_{(b)} < 1 + \sqrt{1 - 1/\rho} \ .$$

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

Theorem 10 *Let $\rho = c_{max}/c_{min} \geq 1$. In order to have competitiveness, it is sufficient to set $s = s_{(a)} = \rho$ if $\rho \in [1, \varphi]$, and $s = s^+_{(b)} = 1 + \sqrt{1 - 1/\rho}$ if $\rho > \varphi$, where $\varphi = \frac{1+\sqrt{5}}{2}$ is the golden ratio.*

Proof: As mentioned before, a sufficient condition for competitiveness is $s \geq \min\{s_{(a)}, s^+_{(b)}\}$. Using calculus is it easy to verify that $s(a) = \rho \leq s^+_{(b)}$ if $\rho \leq \varphi$. ■