Online Distributed Scheduling on a Fault-prone Parallel System

Elli Zavou *1,2 and Antonio Fernández Anta1
1IMDEA Networks Institute, 28918, Leganés (Madrid), Spain
2Universidad Carlos III de Madrid, Madrid, Spain

Abstract

We consider a parallel system of $m$ identical machines prone to unpredictable crashes and restarts, trying to cope with the continuous arrival of tasks to be executed. Tasks have different computational requirements (i.e., processing time or size). The flow of tasks, their size, and the crash and restart of the machines are assumed to be controlled by an adversary. Then, we focus on the study of online distributed algorithms for the efficient scheduling of the tasks. We use competitive analysis, considering as efficiency metric the completed-load, i.e., the aggregated size of the completed tasks.

We first present optimal completed-load competitiveness algorithms when the number of different task sizes that can be injected by the adversary is bounded. (It is known that, if it is not bounded, competitiveness is not achievable.) We first consider only two different task sizes, and then proceed to $k$ different ones, showing in both cases that the optimal completed-load competitiveness can be achieved.

Then, we consider the possibility of having some form of resource augmentation, allowing the scheduling algorithm to run with a speedup $s \geq 1$. In this case, we show that the competitiveness of all work-conserving scheduling algorithms can be increased by using a large enough speedup.

Keywords: Scheduling, Parallel Computation, Non-uniform Tasks, Failures, Competitiveness, Online Algorithms

1 Introduction

With the widespread use of cloud computing (which is essentially equivalent to computing in large scale data centers) and big data processing, parallel computing is taking new forms. Parallelism appears as the execution of lightly coupled tasks (or jobs), like the map or reduce tasks of a map-reduce computation, in a collection of decoupled processors (or cores). The large scale of both size and time of these types of computation, has two important consequences. First, it makes it highly likely that processors will fail during the computation (failures are the norm, not the exception [9, 28, 17, 6]), and hence recovery mechanisms must be an intrinsic part of the task scheduler. Second, it is unlikely that the information about all the tasks to be scheduled is available at the initial time of the computation, which means that the task scheduler must make online decisions. These two aspects make most prior work on task scheduling on parallel machines not applicable in these new environments (e.g., [10, 15, 16, 22, 25, 26]). For instance, some works tackle the issue of dynamic job arrivals but do not consider failures (e.g., [5]), others consider failures but assume knowledge of all the jobs a priori (e.g., [18, 21]), finally, some others consider energy efficiency issues but not machine failures, or only jobs of the same computational demand (e.g., [8]).

In this paper, we explore the scheduling of tasks in a parallel system like the one depicted in Figure 1 (which is similar to the one introduced in [3]). The system has $m$ identical machines1 prone to crashes and restarts (controlled by an adversary). Independent idempotent tasks with different computational demands

---

*elli.zavou@imdea.org

1We use the terms machine and processor interchangeably.
Figure 1: The computing setting considered, with the $m$ homogeneous machines, the shared Repository, and the three operations; inject, for the dynamic task arrivals from the users, get, for the machines to obtain the set of pending tasks, and inform for the repository to update the set of pending tasks.

(e.g., in terms of processing time) arrive into the system, to be executed by any of the machines. In order to remove a single point of failure, the system considered has no central scheduler. Instead (see Fig. 1), each task that arrives is held in a repository until some machine executes it and reports this fact. Hence, the scheduling is done in a distributed way by the machines. The objective is to design efficient distributed scheduling algorithms that run at the machines. In summary, the characteristics of the parallel system considered here are (1) continuous task arrivals, (2) machine failures and restarts, and (3) distributed scheduling. We measure the performance of an algorithm by its completed-load (i.e., the total size of the completed tasks) competitiveness.

One additional aspect of cloud computing is its impact in terms of energy consumption. It is known that the power consumed by a processor grows with its processing speed (in cycles per second) \[4\]. In our model we introduce this in the form of a speedup \[s \geq 1\], that can be used to allow processors to run faster than the baseline. This is a form of resource augmentation. We show that this resource augmentation, increasing the speed of the machines, improves the competitiveness of the system.

**Related Work**  Probably, the most important research line related to this work is the study of machine scheduling with availability constraints (e.g., \[25, 15, 14, 18\]). One of the most important outcomes of this line is the necessity of algorithms that take into account unexpected machine breakdowns. Most works allow preemptive scheduling \[15, 16\] and show optimality only for nearly online algorithms (for example, algorithms that need to know the time of the next job arrival or machine availability). Among these works some also consider energy issues, and use speed-scaling to tune the power consumption of the processors (e.g., \[1, 7, 8\]).

The work of Georgiou and Kowalski \[14\] was the one that initiated our study. They looked at a rather different setting, consisting of a cooperative computing system of $m$ message-passing processors prone to crashes and restarts, having to collaborate in order to complete dynamically injected tasks. For the efficiency of the system, they performed competitive analysis, focusing on the maximum number of pending tasks. They proved competitiveness with unit-length tasks, and showed that if tasks have different lengths competitiveness cannot be achieved.

In \[3\] we looked at a setting similar to the one used here, with $m$ machines, introduced the term of speedup, representing the resource augmentation required, and showed competitiveness in terms of pending load (sum of sizes of pending tasks). More precisely, tasks of at least two different sizes were considered, and it was found that the threshold for competitiveness was defined by these two conditions: (a) $s < \rho$ and (b) $s < 1 + \gamma/\rho$, where $\rho$ is the ratio between the largest and smallest task and $\gamma$ is a parameter that depends on $\rho$ and $s$ (see \[3\] for details). If both conditions hold then no deterministic algorithm is
| Speedup | 1 - machine | m - machines |
|---------|-------------|-------------|
| $s = 1$ | $C(\text{ALG}_W) = 0$, any task size | $C(\text{ALG}_W) \leq \frac{\bar{p}}{p+\bar{p}} \approx 1/2$, two task sizes | [12] |
|         | $C(\text{ALG}_W) = \frac{\bar{p}}{p+\bar{p}} \approx 1/2$, two task sizes | $C(\rho\text{-Preamble}) \geq \frac{\bar{p}}{p+\bar{p}} \approx 1/2$, two task sizes | [Th. 1] |
|         | $C(\text{Greedy}) \geq \frac{\bar{p}}{p+\bar{p}} \approx 1/2$, two task sizes | $C(\text{Greedy}) \geq \frac{\bar{p}}{p+\bar{p}} \approx 1/2$, two task sizes | [19] |
| $s \geq \rho$ | $C(\text{ALG}_W) \geq 1/\rho$, [12] | $C(\text{ALG}_W) \geq 1/\rho$, [12] |
| $s \geq \max\{\rho, 2\}$ | $C(\text{LIS}) \geq 1$, [12] | $C(\text{m-LIS}) < 1$, when $m = s = \rho = 2$ | [Th. 6] |
| $s \geq 1 + \rho$ | $C(\text{ALG}_W) \geq 1$, [12] | $C(\text{ALG}_W) \geq 1$, [12] |

Table 1: Negative (upper bounds) and positive (lower bounds) results on the completed-load competitiveness. ALG$_W$ is any work conserving algorithm. Note that the negative results hold for both 1-machine and $m$-machines. Recall that $\rho = \frac{\pi_{\min}}{\pi_{\max}}$ and $\bar{\rho} = \lfloor \rho \rfloor$.

Contributions As mentioned, in this work, we consider a setting with $m$ machines prone to crashes and restarts, controlled by an adversary (to model worst-case scenarios), and a shared repository (an entity that
provides the service by which the clients of the system submit the tasks to be executed and notifies them of their completion – see Fig. 1. Note that the shared repository is not a scheduler, since it does not make any decisions on the execution of the tasks. It is basically a passive storage that behaves as an interface between the clients that generate tasks and the machines. It also allows the machines to maintain information about the tasks that have not been executed yet.

Tasks arrive in the system continuously and have different computational demands (which is their size). We assume that each task \( \tau \) has size \( \pi(\tau) \in [\pi_{\min}, \pi_{\max}] \), where \( \pi_{\min} \) and \( \pi_{\max} \) correspond to the smallest and largest possible values respectively, and that \( \pi(\tau) \) only becomes known at the moment of the task arrival. Tasks are held in the shared repository, which is later accessed by the machines in order to decide which task to execute next. Note, then, that the machines’ decisions are taken in a distributed manner and without any communication between them. When a task is completed, the corresponding machine informs the shared repository (which in turn notifies the client).

As mentioned, we consider the possibility of having resource augmentation in the form of speedup \( s \geq 1 \) [20] [2] (i.e., we increase the computational speed of the machines) in order to cope with the performance cutback from the machine failures (crashes) and restarts, as well as the lack of information for the future task arrivals. More precisely, we consider uniform speedup \( s \geq 1 \) for all the machines, under which a task \( \tau \) is executed \( s \) times faster, i.e., in time \( \pi(\tau)/s \).

Since the scheduling decisions must be made in a continuous manner and without future knowledge (neither of the task arrivals nor of the machine crashes and restarts), we study the problem as an online scheduling problem [24] [3] [11]. Hence, for the evaluation of the different algorithms proposed, we use competitive analysis [27], measuring the total completed load of the system: the sum of sizes of the completed tasks. More precisely, an algorithm is considered \( \alpha \)-completed-load competitive, also expressed as \( C(\text{ALG}) = \alpha \), with speedup \( s \) if under any adversarial behavior its completed-load complexity is at least \( \alpha \) times the completed-load complexity of any algorithm \( X \), running with no speedup and the same adversarial behavior. Fully detailed specifications of the model used are given in Section 2.

In Table 1 we summarize our results, including also some relevant results found in previous works. Note that the upper bounds found for the case of one machine, hold directly for the case of \( m \) machines, since the adversary can simply crash all machines except one, and force the corresponding adversarial scenarios to occur. However, the positive results (lower bounds) do not necessarily transfer from 1 to \( m \) machines.

The upper part of Table 1 presents the results obtained in Section 3 when the machines run without speedup (i.e. \( s = 1 \)). As can be seen, for this case we present algorithms that achieve optimal completed-load competitiveness, in all the cases. Observe that, in [12], it was shown for one machine that no work-conserving algorithm\(^2\) can achieve competitiveness if there is an arbitrary number of different task sizes. This, being a negative result, also holds for the case of \( m \)-machines. Then, we give three work-conserving algorithms, focusing on the cases of two task sizes and bounded number of task sizes, with and without pairwise divisibility; a property that holds between each pair of task sizes (explained further in Section 3). For the three cases, algorithms \( \rho m \)-Preamble, \( k \)-Amortized, and \( M k \)-Amortized respectively, achieve optimal competitiveness, matching the upper bound shown in [11]. These algorithms are non-trivial generalization of algorithms proposed in [11] and [19].

The lower part of Table 1 presents the results obtained in Section 4 for systems running with speedup \( s > 1 \). The first interesting observation from these results is that, contrary to intuition, to move from one machine to multiple machines it is not enough to complement an algorithm that works for \( m = 1 \) with a mechanism that prevents redundant execution of tasks when \( m > 1 \). This is shown in the case of \( s \geq \max\{\rho, 2\} \) with the algorithm \( m \)-LIS, proposed in [3], which is the natural adaptation of LIS to multiple machines. As observed, while LIS guarantees 1-completed-load competitiveness in one machine, \( m \)-LIS cannot achieve that level of competitiveness even with 2 machines. Fortunately, as shown, we have been able to generalize two important general positive results obtained for work-conserving algorithms in one machine to multiple machines.

\(^2\)An algorithm is work conserving if it does not allow a machine to be idle if there are tasks to be executed in the repository.
2 Model and Definitions

The parallel system considered has \( m \) identical machines, prone to crashes and restarts, with unique ids in the set \( \{0, 1, 2, \ldots, m-1\} \). Please see also Fig. 1 for the graphical representation of the system. As mentioned, the machines have access to a shared repository. The repository supports three essential operations: inject, get, and inform. The inject operation is executed by a client of the system to add a new task to the current set of tasks to be executed. We assume here that this operation is controlled by an adversary (as will be further discussed below). Operations get and inform are executed by the machines. A machine uses the get operation to obtain the set of pending tasks, i.e., the tasks that are in the repository because they were injected and no machine has notified their completion yet. For simplicity, we assume that the get operation is blocking, i.e., if the repository is empty when executed, it waits until some new task is before returning (the set of newly injected tasks). A machine then executes an inform operation when it has completed the task scheduled, notifying the repository about its completion. Then the repository removes immediately this task. We assume that the execution of these operations is instantaneous (takes negligible time), except a get operation that blocks.

We consider machines running in processing cycles, controlled by the scheduling algorithm considered. Each cycle, starts with a get operation, a task execution, and an inform operation (if the task is completed). Since the repository operations (get and inform) are instantaneous, a processing cycle lasts the time needed for the scheduled task to be completed. We assume that machines run with a speedup \( s \geq 1 \) \( (s = 0 \) means no speedup). Then, a processing cycle lasts a time equal to the size of the task divided by the speedup \( s \). When a machine crashes, the cycle is interrupted and the progress in the task execution is lost. If the machine later recovers, it starts a new cycle.

**Event ordering** Since the injection of tasks by clients, the get operations, and the notification via inform operations of task completion by the machines may occur simultaneously, we define the following order among these events. We assume that in an instant \( t \) the inform operations occur first, then the injections, and finally the get operations. Hence, a get operation executed at time \( t \) will include the tasks injected at time \( t \) but not the ones completed at that time.

**Tasks** As already explained, computational tasks are injected to the system by the clients, with the inject operation at the repository. We assume that this operation is controlled by an arrival pattern \( A \) (a sequence of task injections) defined by an omniscient adversary. Each task \( \tau \) has an arrival time \( a(\tau) \) and a size \( \pi(\tau) \), which is the time required to complete the task without speedup. The task attributes are only known at the time of its injection. We use the term \( \pi \)-task to refer to a task of size \( \pi \in [\pi_{\min}, \pi_{\max}] \). The values \( \pi_{\min} \) and \( \pi_{\max} \) are the smallest and largest possible task sizes, and are usually assumed to be known by the scheduling algorithm. We also define parameter \( \rho = \frac{\pi_{\max}}{\pi_{\min}} \) to be the ratio between the largest and smallest task sizes.

We assume that tasks are atomic in the sense that not executing one completely due to a crash implies that it has to be executed again from the start. On the other hand, we assume the tasks to be idempotent \([13]\), which means that executing the same task more than once has the same effect as executing it only once.

**Machine Crashes and Restarts** For the machine crashes and restart, we consider an omniscient adversary, which is the same entity responsible for the task injections at the repository described above. This means that the adversary is expected to coordinate injections, crashes, and recoveries. In an execution of the system, the adversary defines an error pattern \( E \), which is a list of crash and restart events, each associated with the time it occurs (e.g., \( \text{crash}(t, p) \) is the event that that machine \( p \) is crashed at time \( t \)). We consider a machine \( p \) being alive in time interval \( I = [t, t'] \), if it is operational at time \( t \) and does not crash at any time \( t'' \leq t' \).

**Definition 1** An adversarial pattern is a combination of arrival and error patterns \( A \) and \( E \). It is admissible only when at all time instants there is at least one machine alive. In our work we only consider admissible adversarial patterns.
Notation  We consider it useful to provide all extensively used notation. Since it is essential to keep track of injected, completed and pending tasks at each time instant in an execution, we introduce sets $I_t(A)$, $N_t^s(X,A,E)$ and $Q_t^s(X,A,E)$, where $X$ is an algorithm, $A$ and $E$ the arrival and error patterns respectively, $t$ the time instant under consideration and $s$ the speedup of the machines. $I_t(A)$ represents the set of injected tasks within the interval $[0,t]$, $N_t^s(X,A,E)$ the set of completed tasks within $[0,t]$ and $Q_t^s(X,A,E)$ the set of pending tasks at time instant $t$. As implied by the event ordering defined above, $Q_t^s(X,A,E)$ contains the tasks that were injected by time $t$ inclusively, but not the ones completed before and up to time $t$. Observe that $I_t(A) = N_t^s(X,A,E) \cup Q_t^s(X,A,E)$ and that set $I$ depends only on the arrival pattern $A$, while sets $N$ and $Q$ also depend on the error pattern $E$, the algorithm run by the scheduler, $X$, and the speedup of the machine, $s$. For simplicity, we omit the superscript $s$ in further sections of the paper. However, the appropriate speedup in each case is clearly stated at all times.

We use $L_\pi$ to refer to the subset of $Q_t^\pi(X,A,E)$ that includes only pending tasks of size $\pi$, and we assume an ascending order of tasks in each queue $L_\pi$, according to their arrival time. To simplify the presentation of the algorithms, in a list of pending tasks we number them starting with 0. Then, for instance, the tasks in $L_\pi$ are numbered from 0 to $|L_\pi|$. What is more, we will use notation $|L_\pi(X,t)|$ to refer to the number of $\pi$-tasks pending in the execution of $X$ at time $t$. In a similar way, we use notation $|N_\pi(X,t)|$ to denote the number of completed tasks by machine $p$ in the execution of algorithm $X$ at time $t$.

Finally, we include some definition that originally appeared in [3] and will be used in the rest of the paper.

Definition 2 ([3]) An algorithm is of type GroupLIS, if all the following hold:

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

Efficiency Measures We evaluate our algorithms with the completed load measure. Given an algorithm $\text{ALG}$ running with speedup $s \geq 1$, and adversarial arrival and error patterns $A$ and $E$ respectively, we look at time $t \geq 0$ of the execution and focus on the completed load complexity. This means, that we look at the sum of sizes of the completed tasks up to time instant $t$:

$$C_t^\pi(\text{ALG},A,E) = \sum_{\tau \in N_t^s(\text{ALG},A,E)} \pi(\tau)$$

Finding the algorithm, in other words computing the schedule, that maximizes the measure offline (having the knowledge of patterns $A$ and $E$) is an NP-hard problem [3] [11].

We will also be using a slightly changed notation $C^X(t,\pi)$ – resp., $P^X(t,\pi)$ – to denote the completed load – resp., pending load – of size $\pi$ at time instant $t$ in the execution of algorithm $X$. Similarly, $C^X(t,\pi)$ – resp., $P^X(t,\pi)$ – refers to the completed load – resp., pending load – of size smaller than $\pi$ at time instant $t$ in the execution of $X$.

As already mentioned, since the system is dynamic in respect to the task arrivals and machine crashes and restarts, we view the scheduling problem of task as an online one, and pursue competitive analysis. Specifically, considering any time $t$ of an execution, any combination of arrival and error patterns, $A$ and $E$, and any algorithm $X$ designed to solve the scheduling problem, the completed load competitiveness of an algorithm $\text{ALG}$ that runs with speedup $s \geq 1$ is defined as follows:

Algorithm $\text{ALG}$ is $\alpha$-completed-load competitive if $\forall t, X, A,E, C_t^\pi(\text{ALG},A,E) \geq \alpha \cdot C_t^\pi(X,A,E) + \Delta_C$, where parameter $\Delta_C$ does not depend on $t$, $X$, $A$ or $E$, and $\alpha$ is the completed-load competitive ratio, which we denote by $\mathcal{C}(\text{ALG})$. What is more, $\alpha$ is also independent of $t$, $X$, $A$ and $E$, but it may depend on system parameters like $\pi_{\text{min}}, \pi_{\text{max}}, m$ or $s$, which are not considered as inputs of the problem; they are fixed and given upfront. The input of the problem is formed only by the adversarial arrival and error patterns $A$. 

Algorithm 1 $\rho m$-Preamble (for machine $p$)

Parameters: $m, \pi_{\text{min}}, \pi_{\text{max}}$

Upon awaking or restart
   Get $L_{\pi_{\text{min}}}$ and $L_{\pi_{\text{max}}}$, from the Repository;
   $\text{preamble} \leftarrow \text{FALSE}$                               //Reset preamble status
   $c \leftarrow 0$;                                                    //Reset counter
   Calculate $\overline{\rho} \leftarrow \left\lfloor \frac{\pi_{\text{max}}}{\pi_{\text{min}}} \right\rfloor$
   If $|L_{\pi_{\text{min}}}| \geq \overline{\rho} \cdot m^2$ then
      $\text{preamble} \leftarrow \text{TRUE}$;

Repeat
   Get the queues of pending tasks, $L_{\pi_{\text{min}}}$ and $L_{\pi_{\text{max}}}$, from the Repository;
   Sort $L_{\pi_{\text{min}}}$ and $L_{\pi_{\text{max}}}$ by arrival time (ascending);
   If $\text{preamble} = \text{TRUE} \land (c < \overline{\rho})$ then
      execute task at position $p \cdot m$ in $L_{\pi_{\text{min}}}$;
      $c \leftarrow c + 1$;
   else
      If $|L_{\pi_{\text{max}}}| \geq m^2$ then
         execute task at position $p \cdot m$ in $L_{\pi_{\text{max}}}$;
      else if $|L_{\pi_{\text{min}}}| \geq m^2$ then
         execute task at position $p \cdot m$ in $L_{\pi_{\text{min}}}$;
      else if $L_{\pi_{\text{max}}} \neq \emptyset$ then
         execute task at position $(p \cdot m) \mod |L_{\pi_{\text{max}}}|$ in $L_{\pi_{\text{max}}}$;
      else if $L_{\pi_{\text{min}}} \neq \emptyset$ then
         execute task at position $(p \cdot m) \mod |L_{\pi_{\text{min}}}|$ in $L_{\pi_{\text{min}}}$;
   Inform the Repository for the task completion;

and $E$. Finally, let us clarify that the number of machines $m$ is fixed for a given execution, and that the algorithm used may take it into consideration; hence different $m$ may result to different performance of the same algorithm, affecting additive term in the competitiveness. The same holds for $\pi_{\text{min}}, \pi_{\text{max}}$ and $s$.

3 No Speedup

Let us start with the section in which machines have no speedup, i.e., $s = 1$. We aim to show that the upper bound of completed-load competitiveness shown in [11] can be achieved by some online algorithms in the distributed setting of $m$ machines. In particular, we propose three scheduling algorithms, $\rho m$-Preamble, $k$-Amortized and $Mk$-Amortized, and analyze their performance under worst-case arrival and error patterns $A$ and $E$, showing that the upper bound of completed-load competitiveness is guaranteed.

3.1 Two Task Sizes

Let us start with the first algorithm, $\rho m$-Preamble, that runs on the $m$ machines of the system and considers only two different task sizes, $\pi_{\text{min}}$ and $\pi_{\text{max}}$ (see the algorithm’s pseudocode Alg. [1]).

Algorithm description. Upon awaking or restart, machine $p$ reads the two queues of pending tasks from the Repository, $L_{\pi_{\text{min}}}$ and $L_{\pi_{\text{max}}}$, and applies ascending sort by their arrival time, such that the earliest injected task is at position 1 of the queue. It then calculates parameter $\overline{\rho} = |\rho|$ and along with parameter preamble decides which is the next task to be scheduled, avoiding redundancy when enough tasks are pending. More precisely, at each decision time, if there are at least $\overline{\rho} \cdot m^2$ tasks of size $\pi_{\text{min}}$ and preamble = TRUE, the machine attempts to complete $\overline{\rho} \pi_{\text{min}}$-tasks before continuing with a non-redundant
version of the Largest Size (LS) scheduling approach. Let us explain further: after the preamble is completed by the machine (if there was enough time and the machine did not crash), it gives priority to the largest tasks, given that it has enough of them, so that redundancy is avoided (see exact conditions in Algorithm 1). Note that if there are at least $m^2 \pi_{\text{min}}$-tasks and/or at least $m^2 \pi_{\text{max}}$-tasks, each machine will complete a different task of the same size. Hence, if there are not enough $\pi_{\text{max}}$-task but there are enough $\pi_{\text{min}}$ ones, it will schedule $\pi_{\text{min}}$-tasks instead of risking redundant executions with scheduling $\pi_{\text{max}}$ ones.

Observe that algorithm $\rho m$-Preamble belongs to the class of scheduling algorithms named GroupLIS (see definition in Section 2), which was initially defined by Fernández et al. in [3]. They showed that the algorithms in this class, considering speedup $s \geq 1$, do not execute the same task twice, as long as there are enough pending tasks (i.e., $\geq m^2$), and thus we show that the same holds for $\rho m$-Preamble. Let us start with the next lemma, that corresponds to the adaption of Lemma 8 in [3] for the case of no speedup.

**Definition 3 ([3])** A full task execution of a task $\tau$ is the interval $[t, t']$, during which a machine $p$ schedules $\tau$ at time $t$ and reports its completion to the repository at $t'$, without stopping its execution within the interval $[t, t']$.

**Lemma 1** For an algorithm ALG of type GroupLIS and a time interval $T$ in which a queue $L_\pi$ has at least $m^2$ pending tasks, any two full task executions (in $T$) by different machines, are executions of different tasks; i.e., the full executions of tasks $\tau_1, \tau_2 \in L_\pi$ by machines $p_1$ and $p_2$ respectively, must have $\tau_1 \neq \tau_2$.

For the ease of presentation, let us use letter $A$ to refer to our algorithm, $\rho m$-Preamble, in its analysis. Let us also define the following two intervals, during which there are sufficient tasks pending in order to guarantee non-redundant task executions (from Lemma 1 we make the observation that follows):

- $T^+$: an interval where $|L_{\pi_{\text{max}}}(A, t)| \geq m^2$, $\forall t \in T^+$
- $T^-$: an interval where $|L_{\pi_{\text{min}}}(A, t)| \geq m^2$, $\forall t \in T^-$

**Observation 1** All full executions of $\pi_{\text{max}}$-tasks in the execution of algorithm $\rho m$-Preamble within any interval $T^+$ appear exactly once. Similarly, all full executions of $\pi_{\text{min}}$-tasks in the execution of $\rho m$-Preamble within an interval $T^-$ appear exactly once.

Note now, that there are two possible types for the whole execution of Algorithm $\rho m$-Preamble:

(a) $\forall t, \exists t' > t$ such that

$|L_{\pi_{\text{min}}}(A, t')| < \overline{p}m^2 \land |L_{\pi_{\text{max}}}(A, t')| < m^2$.

(b) $\exists t'$ such that $\forall t' > t$

$|L_{\pi_{\text{min}}}(A, t')| \geq \overline{p}m^2 \lor |L_{\pi_{\text{max}}}(A, t')| \geq m^2$.

In the first case, when an execution is of type (a), there will always be a time $t'$ after the current time instant $t$, at which the queue $L_{\pi_{\text{min}}}$ has less than $\overline{p}m^2$ tasks and the queue $L_{\pi_{\text{max}}}$ less than $m^2$ tasks. In this case, non-redundancy cannot be guaranteed.

In the second case, when an execution is of type (b), after time $t$ the queue of pending tasks will never become empty, it will instead have enough pending tasks in order to guarantee non-redundancy at all times. The execution after time instant $t$ can be described by a sequence of intervals, say $T_i$, where $i$ simply denotes their sequence. They are $T^+$ and/or $T^-$ intervals. More precisely, they belong to one of the following types:

1. $|L_{\pi_{\text{min}}}(A, t^*)| \geq \overline{p}m^2 \land |L_{\pi_{\text{max}}}(A, t^*)| \geq m^2$, $\forall t^* \in T$

   In this case, a machine following algorithm $\rho m$-Preamble will schedule $\overline{p} \pi_{\text{min}}$-tasks, followed by continuously scheduled $\pi_{\text{max}}$-tasks, until a time instant $t$ where either $|L_{\pi_{\text{min}}}(A, t)| < \overline{p}m^2$ or $|L_{\pi_{\text{max}}}(A, t)| < m^2$, thus one of the next two types of period will follow.

Note that if there are at least $m^2 \pi_{\text{min}}$-tasks and/or at least $m^2 \pi_{\text{max}}$-tasks, each machine will complete a different task of the same size. Hence, if there are not enough $\pi_{\text{max}}$-task but there are enough $\pi_{\text{min}}$ ones, it will schedule $\pi_{\text{min}}$-tasks instead of risking redundant executions with scheduling $\pi_{\text{max}}$ ones.

Observe that algorithm $\rho m$-Preamble belongs to the class of scheduling algorithms named GroupLIS (see definition in Section 2), which was initially defined by Fernández et al. in [3]. They showed that the algorithms in this class, considering speedup $s \geq 1$, do not execute the same task twice, as long as there are enough pending tasks (i.e., $\geq m^2$), and thus we show that the same holds for $\rho m$-Preamble. Let us start with the next lemma, that corresponds to the adaption of Lemma 8 in [3] for the case of no speedup.

**Definition 3 ([3])** A full task execution of a task $\tau$ is the interval $[t, t']$, during which a machine $p$ schedules $\tau$ at time $t$ and reports its completion to the repository at $t'$, without stopping its execution within the interval $[t, t']$.

**Lemma 1** For an algorithm ALG of type GroupLIS and a time interval $T$ in which a queue $L_\pi$ has at least $m^2$ pending tasks, any two full task executions (in $T$) by different machines, are executions of different tasks; i.e., the full executions of tasks $\tau_1, \tau_2 \in L_\pi$ by machines $p_1$ and $p_2$ respectively, must have $\tau_1 \neq \tau_2$.

For the ease of presentation, let us use letter $A$ to refer to our algorithm, $\rho m$-Preamble, in its analysis. Let us also define the following two intervals, during which there are sufficient tasks pending in order to guarantee non-redundant task executions (from Lemma 1 we make the observation that follows):

- $T^+$: an interval where $|L_{\pi_{\text{max}}}(A, t)| \geq m^2$, $\forall t \in T^+$
- $T^-$: an interval where $|L_{\pi_{\text{min}}}(A, t)| \geq m^2$, $\forall t \in T^-$

**Observation 1** All full executions of $\pi_{\text{max}}$-tasks in the execution of algorithm $\rho m$-Preamble within any interval $T^+$ appear exactly once. Similarly, all full executions of $\pi_{\text{min}}$-tasks in the execution of $\rho m$-Preamble within an interval $T^-$ appear exactly once.

Note now, that there are two possible types for the whole execution of Algorithm $\rho m$-Preamble:

(a) $\forall t, \exists t' > t$ such that

$|L_{\pi_{\text{min}}}(A, t')| < \overline{p}m^2 \land |L_{\pi_{\text{max}}}(A, t')| < m^2$.

(b) $\exists t'$ such that $\forall t' > t$

$|L_{\pi_{\text{min}}}(A, t')| \geq \overline{p}m^2 \lor |L_{\pi_{\text{max}}}(A, t')| \geq m^2$.

In the first case, when an execution is of type (a), there will always be a time $t'$ after the current time instant $t$, at which the queue $L_{\pi_{\text{min}}}$ has less than $\overline{p}m^2$ tasks and the queue $L_{\pi_{\text{max}}}$ less than $m^2$ tasks. In this case, non-redundancy cannot be guaranteed.

In the second case, when an execution is of type (b), after time $t$ the queue of pending tasks will never become empty, it will instead have enough pending tasks in order to guarantee non-redundancy at all times. The execution after time instant $t$ can be described by a sequence of intervals, say $T_i$, where $i$ simply denotes their sequence. They are $T^+$ and/or $T^-$ intervals. More precisely, they belong to one of the following types:

1. $|L_{\pi_{\text{min}}}(A, t^*)| \geq \overline{p}m^2 \land |L_{\pi_{\text{max}}}(A, t^*)| \geq m^2$, $\forall t^* \in T$

   In this case, a machine following algorithm $\rho m$-Preamble will schedule $\overline{p} \pi_{\text{min}}$-tasks, followed by continuously scheduled $\pi_{\text{max}}$-tasks, until a time instant $t$ where either $|L_{\pi_{\text{min}}}(A, t)| < \overline{p}m^2$ or $|L_{\pi_{\text{max}}}(A, t)| < m^2$, thus one of the next two types of period will follow.
In this case, a machine following algorithm $\rho m$-Preamble will continuously schedule $\pi_{min}$-tasks until the queues are such that one of the other two types of periods follow.

(3) $|L_{\pi_{min}}(A, t^*)| < \rho m^2 \cap |L_{\pi_{max}}(A, t^*)| > m^2, \forall t^* \in T$

In the third case, machines following the $\rho m$-Preamble continuously schedule $\pi_{max}$-tasks, until the queues become such that one the previous two types of periods follow.

Lemma 2 For executions of type (a), where $\forall t, \exists t' > t$ s.t. $|L_{\pi_{min}}(A, t')| < \rho m^2 \cap |L_{\pi_{max}}(A, t')| < m^2$ holds, the completed-load competitive ratio of algorithm $\rho m$-Preamble goes to 1, i.e., $C(\rho m$-Preamble) = 1.

Proof: First, let us fix a pair of arrival and error patterns, such that executions of case (a) occur. We focus on time instant $t'$ from the definition. Observe that at time $t'$, the total pending load of algorithm $\rho m$-Preamble is less than $\rho m^2 \pi_{min} + m^2 \pi_{max}$, while the total pending load of $X$ is at least zero. Let us overload the notation of the set of injected tasks up to time $t'$ such as to represent the total injected load up to that time; $I t'$ will now represent the sum of sizes of all injected tasks up to time $t'$. Then, at time instant $t'$ the completed load ratio is

$$C(t') = \frac{C'(\rho m$-Preamble)}{C'(X)} = \frac{I t' - (|L_{\pi_{min}}(A, t')| \pi_{min} + |L_{\pi_{max}}(A, t')| \pi_{max})}{I t' - (|L_{\pi_{min}}(X, t')| \pi_{min} + |L_{\pi_{max}}(X, t')| \pi_{max})}$$

which leads to a completed-load competitive ratio of 1 as time goes to infinity; the total injected load goes to infinity as well:

$$C(\rho m$-Preamble) = \lim_{t' \to \infty} C(t') = \lim_{t' \to \infty} \left(1 - \frac{\rho m^2 \pi_{min} + m^2 \pi_{max}}{I t'}\right) = 1.$$ 

This completes the proof of the completed load competitiveness for executions of type (a) as claimed. 

The next lemma follows mostly the idea of analysis of algorithm SL-Preamble, presented in [11], for the case of packet scheduling over one communication link. The complete proof is included in the Appendix.

Lemma 3 For executions of type (b), where $\exists t, \forall t' > t$, s.t. $|L_{\pi_{min}}(A, t')| \geq \rho m^2 \cap |L_{\pi_{max}}(A, t')| \geq m^2$ holds, the completed load competitive ratio of algorithm $\rho m$-Preamble is $C(\rho m$-Preamble) $\geq \frac{\rho}{\rho + 1}$.

From Lemmas 2 and 3, that analyze the two types of executions, (a) and (b) respectively, we have the lower bound for the completed-load competitiveness of $\rho m$-Preamble, given by the following theorem.

Theorem 1 When algorithm $\rho m$-Preamble runs without speedup ($s = 1$) under any arrival and error patterns $A$ and $E$, it has a completed-load competitive ratio $C(\rho m$-Preamble, $A, E) \geq \frac{\rho}{\rho + 1}$.

3.2 Finite Task Sizes – Pairwise Divisible

We now move to the case of $k > 2$ different task sizes. Let us denote them by $\pi_{min} = \pi_1 < \pi_2 < \cdots < \pi_k = \pi_{max}$. We assume that each ratio $\rho_{i,j} = \pi_i / \pi_j$ is an integer for any $1 \leq j < i \leq k$, a property of the task sizes called pairwise divisibility.

Inspired by the work of Jurdzinski et al. [19], we propose algorithm $k$-Amortized, that uses this property to schedule the tasks among the $m$ machines of the system, and analyze its completed-load competitiveness when run without speedup (see the algorithm’s pseudocode in Alg. 2 and 3). The algorithm follows the Shortest Size (SS) first policy, but subject to some balancing constraints. It is based on scheduling tasks in
Algorithm 2 $k$-Amortized (for machine $p$)

1 Parameters: $m, \{\pi_1, \pi_2, \ldots, \pi_k\}$
2 Upon awaking or restart
3 Repeat
4 Get $L_1$ to $L_k$ from the Repository;
5 While $\pi_k \left\lfloor \frac{|L_k|}{m} \right\rfloor + \sum_{i=1}^{k-1} \pi_i \left\lfloor \frac{|L_i|}{m^2 + m^{i+1}} \right\rfloor < \pi_k$ Do
6 execute task $\pi$ at position $(p \cdot m) \mod |Q|$ in $Q$;
7 Inform the Repository for the task completion;
8 Schedule_Group($k$);

Algorithm 3 Schedule_Group($j$)

1 Parameters: $m, j, \{\pi_1, \pi_2, \ldots, \pi_k\}, \{L_1, L_2, \ldots, L_j\}$
2 If $\sum_{i=1}^{j-1} \pi_i \left\lfloor \frac{|L_i|}{m^2 + m^{i+1}} \right\rfloor \geq \pi_j$ then
3 For $\alpha = 1$ to $\rho_j$ Do
4 Schedule_Group($j-1$);
5 Else
6 execute task $\pi_j$ at position $p \cdot m$ in $L_j$;
7 Inform the Repository for the task completion;
8 Return

Batches (or groups) that balance the length of the next larger task. What is more, it considers redundancy avoidance, demanding enough tasks available before scheduling. An important difference with algorithm $\rho m$-Preamble, apart from the fact that this one considers $k$ different task sizes, is that it continuously schedules bursts of short tasks before going to the next larger task (if the tasks are available and the machines do not crash).

Special Notation & Terminology Before looking at the details of the algorithm, let us introduce some necessary notation and terminology that will be used extensively in this subsection. First, parameter $\rho_{i,j} = \pi_i / \pi_j$, where $1 \leq j < i \leq k$, as already mentioned at the beginning of the subsection, represents the ratio between two task sizes and is considered to be an integer for this algorithm. A special case of this ratio used in the algorithm, is $\rho_i = \pi_i / \pi_{i-1}$, where $i \in [2, k]$; it represents the ratio between two consecutive task sizes.

Definition 4 We define adequate sizes of pending tasks, the task sizes whose pending queues have $\geq m^2$ tasks. More precisely, for size $\pi_k$ to be adequate there must be at least $m^2$ tasks pending in the $L_k$ queue, while for any other $\pi_j$ size, where $j \in [1, k - 1]$, the corresponding pending queue $L_j$ must have at least $m^2 + m^{j+1}$ tasks. (In Lemma 4, we show that this is the necessary number of tasks in order to guarantee the non-redundancy property of the algorithm.)

Definition 5 We define an $i$-group of tasks, where $i \in [1, k]$, being the tasks completed in the execution of a machine under the recursive call to the function Schedule_Group($i$). Note, that an $i$-group has a total size of $\pi_i$, but may be the result of the completion of several smaller tasks.

Definition 6 We consider a machine to be busy at time $t$ if it is either executing some task, it has just completed one, or it is crashed (it is either the time instant that the machine was just crashed, or the machine has been crashed for some time). Observe that an interval from a crash to a restart instant, say $T = [t_c, t_r]$, belongs to the busy interval; during that interval, no algorithm is able to complete any task, hence it does not affect the completed load. Otherwise, it is considered to be idle.

We also consider a machine to be in an $n$-busy interval, say $T = [t_1, t_2]$, where $t_1 < t_2$, satisfying the following properties:
(1) The machine is busy at each time $t \in T$.
(2) The machine does not schedule tasks of size $\pi_i$ for $i > n$ during the interval $T$.
(3) At the beginning of interval $T$, i.e., time instant $t_1$, algorithm $k$-Amortized has at least as many tasks of size $\pi_i$ pending as $X$, for each $i \leq n$. Hence, $P^A(t_1, \pi_i) \geq P^X(t_1, \pi_i)$ for each $i \in [1, n]$.

Finally, note that in the pseudo-code we refer to queue of pending tasks $Q$. Recall that in Section 2 we define $Q$ to be the set of pending tasks. Here, we overload its definition to make this set a unified queue of all $L_i$, sorted in an ascending order according to the task sizes.

**Algorithm description.** After awakening or restart, a machine schedules the task at position $p \cdot m$ of the pending queue $Q$, until the sum of the adequate sizes of the pending tasks is at least $\pi_k$. Following this strategy, the algorithm guarantees the ability to cover a time interval of length $\pi_k$, with non-redundant task executions, if it is not interrupted by a machine crash. Otherwise, being work-conserving, it schedules the task in the position already mentioned, but with no guarantees of non-redundancy. Then, it calls the recursive function $Schedule\_Group(j)$ (starting with $j = k$) which checks whether the sum of adequate sizes of the pending tasks smaller than $\pi_j$ is at least equal to $\pi_j$ (resp., $\pi_k$). If the condition is true, the algorithm makes $\rho_j = \frac{\pi_j}{\pi_j - 1}$ calls to function $Schedule\_Group(j - 1)$ (resp., $Schedule\_Group(k - 1)$) in order to cover the corresponding time interval $\pi_j$ with $\rho_j$ groups of $\pi_{j - 1}$ aggregate size; in other words, $\rho_j (j - 1)$-groups. In the following recursion levels more recursive calls may occur, if there are again enough pending tasks, thus covering the corresponding time intervals by tasks of smaller size each time. Otherwise, when there are not enough tasks pending in a recursion level, a task of the current size, $\pi_j$, is scheduled by the machine and when completed returns to the previous recursion level.

We will now analyze algorithm $k$-Amortized, proving some important properties and showing that its completed-load competitiveness is indeed optimal, i.e., $C(k\text{-Amortized}) \geq \frac{1}{2}$. We start with two lemmas that lead to the non-redundancy property of the algorithm, omitting the case when lines 6 and 7 of Algorithm 2 are executed. Note that in that case, the pending load of the algorithm is bounded, so it does not affect the completed-load competitiveness in the long run.

**Lemma 4** Algorithm $k$-Amortized schedules a $\pi_j$-task (in line 6 of Alg. 3), only when there are at least $m^2$ tasks in the corresponding queue of pending tasks, $L_j$.

**Proof:** Let us start by looking at the algorithm description and its pseudo-code. The first call to schedule some tasks — calling function $Schedule\_Group(k)$ — is done only if enough tasks are pending to cover the $\pi_k$-time without redundancy. A task size is accounted for only when it is adequate; only when there are $\geq m^2$ tasks of that size pending (recall Definition 4).

Then, within the $Schedule\_Group(j)$ function, starting by $j = k$, the algorithm checks whether it can be covered non-redundantly by tasks of smaller size. If it does, it makes a recursive call to the function with parameter $j - 1$, which corresponds to the next smaller task size, $\pi_{j - 1}$. A task $\pi_j$ cannot be covered non-redundantly by smaller tasks when
\[
\sum_{i=1}^{j-1} \pi_i \left| \frac{L_i}{m^2 + m\rho_{i+1}} \right| < \pi_j \quad \text{(see condition of line 2 in Algorithm 3)}.
\]

However, a function call with task size $\pi_j$ means that it was either called by algorithm $k$-Amortized directly (and $j = k$), in which case the condition in line 8 of Alg. 2 does not hold, or it was called by the previous recursion level; by function handling the next bigger task size, $\pi_{j+1}$, in which case the condition in line 2 holds. We will now show that in either case, there are enough tasks of size $\pi_j$ to be scheduled by the system’s machines without executing any of them redundantly.

We consider the function call $Schedule\_Group(j)$, for which the if condition in line 2 does not hold. This implies
\[
\sum_{i=1}^{j-1} \pi_i \left| \frac{L_i}{m^2 + m\rho_{i+1}} \right| < \pi_j.
\] (1)

We then consider and analyze the two cases mentioned above, separately:
Case 1: A previous function call, Schedule_Group\((j + 1)\), in which the if condition in line 2 holds, i.e.,
\[
\sum_{i=1}^{j} \pi_i \left| \frac{|L_i|}{m^2 + m\rho_{i+1}} \right| \geq \pi_{j+1},
\]
implies that
\[
\pi_j \left| \frac{|L_j|}{m^2 + m\rho_{j+1}} \right| + \sum_{i=1}^{j-1} \pi_i \left| \frac{|L_i|}{m^2 + m\rho_{i+1}} \right| \geq \pi_{j+1}
\]
(2)
Combining the two equations 1 and 2 we have the following
\[
\pi_j \left| \frac{|L_j|}{m^2 + m\rho_{j+1}} \right| \geq \pi_{j+1} - \sum_{i=1}^{j-1} \pi_i \left| \frac{|L_i|}{m^2 + m\rho_{i+1}} \right| > \pi_{j+1} - \pi_j > 0
\]
\[
\Rightarrow \left| \frac{|L_j|}{m^2 + m\rho_{j+1}} \right| > 0,
\]
which means that \(|L_j| \geq m^2 + m\rho_{j+1}\).

Case 2: The function call Schedule_Group\((j)\) was actually Schedule_Group\((k)\) and came directly from line 8 of algorithm k-Amortized. Hence, \(\pi_k \left| \frac{|L_k|}{m^2} \right| + \sum_{i=1}^{k-1} \pi_i \left| \frac{|L_i|}{m^2 + m\rho_{i+1}} \right| \geq \pi_k\) holds, which implies that
\[
\pi_k \left| \frac{|L_k|}{m^2} \right| \geq \pi_k - \sum_{i=1}^{k-1} \pi_i \left| \frac{|L_i|}{m^2 + m\rho_{i+1}} \right|
\]
(3)
Replacing \(j = k\) in equation 1, we have
\[
\sum_{i=1}^{k-1} \pi_i \left| \frac{|L_i|}{m^2 + m\rho_{i+1}} \right| < \pi_k,
\]
which combined with equation 3 we can easily see that
\[
\pi_k \left| \frac{|L_k|}{m^2} \right| > 0 \Rightarrow \left| \frac{|L_k|}{m^2} \right| > 0,
\]
which in its turn means that \(|L_k| \geq m^2\).

In both cases, we have seen that there are at least \(m^2\) tasks of size \(\pi_j\) or \(\pi_k\) respectively. However, in case 1 above, there will be \(\rho_{j+1}\) iterations of the recursive function call Schedule_Group\((j)\) of line 4. We must therefore make sure that there are at least \(m^2\) available tasks for all iterations.

Consider for example, the case in which at a time \(t\) all \(m\) machines are in a \((j+1)\)-group execution; following the Schedule_Group\((j + 1)\) function, and having condition of line 2 TRUE. Then, they all start the \(\rho_{j+1}\) iterations of scheduling \(j\)-groups of tasks, calling the recursive function Schedule_Group\((j)\). Consider now, that in all corresponding conditions of line 2, are FALSE. This means, that all \(m\) machines will simultaneously execute one \(\pi_j\)-task in every iteration. Therefore, having \(m^2 + m\rho_{j+1}\) pending tasks of size \(\pi_j\) at the beginning of iterations, will guarantee that in every iteration there are still at least \(m^2\) tasks pending in queue \(L_j\). This completes the proof of the lemma.

Observe now, that algorithm \(\rho\)-m-Preamble is of type GroupLIS; it separates the pending tasks into classes depending on their size, it sorts them with respect to their arrival time, and if a class contains at least \(m^2\) pending tasks, a machine \(p\) schedules the task at position \((p \cdot m)\). Hence, Lemma 1 also holds for it. Hence, combining the two Lemmas 1 and 2 the following property for algorithm k-Amortized follows.

Observation 2 When Algorithm k-Amortized, schedules tasks through its function Schedule_Group (Alg. 3), it never completes the same task more than once. In particular, the same task cannot be simultaneously executed in more than one machines of the system.

Lemma 5 When a task of size \(\pi_j\) is scheduled by k-Amortized, through its function Schedule_Group, say at time instant \(t\), the total size of smaller pending tasks is
\[
P_k(t, < \pi_j) \leq \sum_{i=1}^{j-1} (\pi_j + \pi_i)(m^2 + m\rho_{i+1}).
\]
Proof: When a task of size $\pi_j$ is scheduled by $k$-Amortized in line 6 of Alg. 3, as we have seen also from Lemma 1, the following inequality must hold: 
$$\pi_i \left| \frac{|L_i|}{m^2 + m\rho_{i+1}} \right| < \pi_j.$$ 
This also means that $\forall i \in [1, j-1]$, the following is true:
$$\pi_i \left| \frac{|L_i|}{m^2 + m\rho_{i+1}} \right| < \pi_j \Rightarrow \pi_i \left( \frac{|L_i|}{m^2 + m\rho_{i+1}} - 1 \right) < \pi_j \Rightarrow \pi_i |L_i| < (\pi_j + \pi_i)(m^2 + m\rho_{i+1}).$$

Therefore, the sum of all pending tasks smaller than $\pi_j$ is $P^A(t, < \pi_j) = \sum_{i=1}^{j-1} \pi_i |L_i| \leq \sum_{i=1}^{j-1} (\pi_j + \pi_i)(m^2 + m\rho_{i+1})$ as claimed.

By Observation 2, we have that no task is executed more than once by algorithm $k$-Amortized, when scheduled by its function $\text{Schedule\_Group}$. Hence, we can safely separate the analysis of each machine individually, safely ignoring any task execution by line 6 of Alg. 2. We focus on one machine, say $p$, and then generalizing for all $m$ machines to give the final result. We look at the machine’s $n$-busy intervals, for some $n \leq k$, and prove a completed load competitiveness of $1/2$, provided that $\rho_{i,j} \in \mathbb{N}$ for $1 \leq j < i \leq k$, which is in fact optimal. (The omitted proofs can be found in the Appendix).

Lemma 6 For a machine $p$ that is $n$-busy at a time interval $T$, where $n \leq k$, its total completed load with $k$-Amortized is at least as large as its completed load with $X$ accounting only tasks of size $\geq \pi_n$, in interval $T$, minus $\pi_k$; i.e., $C_p^A(T) \geq C_p^X(T, \geq \pi_n) - \pi_k$.

Lemma 7 For a machine $p$ that is $n$-busy at a time interval $T = [t_1, t_2]$, where $n \leq k$, assume time $t \in T$ be any time when $k$-Amortized starts executing a $\pi_n$-task. Then,
$$2C_p^A([t_1, t]) \geq C_p^X([t_1, t]) + P^X(t, < \pi_n) - \sum_{i=1}^{n-1} (\pi_n + \pi_i)(m^2 + m\rho_{i+1}) - \pi_k.$$

Lemma 8 Let $f_n$ be a function such that $f_1 = \pi_k$ and $f_{i+1} = f_i + \sum_{j=1}^{i} (\pi_{i+1} + \pi_j)(m^2 + m\rho_{j+1}) + \pi_{i+1} + 2\pi_k$. For a machine $p$ that is $n$-busy at a time interval $T$, where $n \leq k$,
$$2C_p^A(T) \geq C_p^X(T) - f_n.$$
3.3 Finite Task Sizes – General

We now look at the case when $\rho_{i,j} = \pi_i / \pi_j \notin \mathbb{N}$. Theorem 2 in [19] shows that the completed load competitive ratio of any scheduling algorithm running without speedup, is at most $\min_{i \in \mathbb{N}} \{ i \pi_{i+1} \}$. This upper bound also holds in the case of multiple machines, since the adversary can force only one of the machines to be alive at all times, while keeping the rest crashed.

However, if algorithm $k$-Amortized is used in this case, the additional advantage of an offline algorithm $X$ from rounding on each recursion level, can accumulate and worsen the competitiveness ratio proven in the previous section. Thus, we must present a modified algorithm, that can reach the upper bound mentioned and hence be optimal.

Let us assume a modification such that instead of executing $\rho_i (i - 1)$-groups on the recursion level $i - 1$, the algorithm keeps sending $(i - 1)$-groups while the completed load of the tasks in the groups completed are less than $\pi_i - \pi_{i-1}$. We can then show the following.

Claim 3.1 The modified algorithm described above has a completed load competitiveness ratio at least

$$\min_{i \in [2,k]} \{ \frac{\rho_{n-1}}{2\rho_{n-1}} \}.$$  

Proof: Looking at Lemmas 3, 7 and 8 and $n$-busy intervals, we use the association of each $\pi_i$-task, for $i \geq n$, completed by $X$, with $\rho_{i,n}$ $n$-groups completed by $k$-Amortized at the same interval, hence covering the $\pi_i$ completed load. For the modified algorithm, a task $\pi_i$ completed by $X$ would correspond to a group of tasks of aggregate size at least $\pi_i - \pi_{i-1}$, which would translate to the following three equations:

$$\frac{\rho_n}{\rho_{n-1}} C^X(T) \geq C^X(T, \geq \pi_n) - \pi_k$$

$$\left(i + \frac{\rho_n}{\rho_{n-1}}\right) C^A([t_1,t]) \geq C^X([t_1,t]) + P^X(t, < \pi_n) - \sum_{i=1}^{n} (\pi_{n+1} + \pi_i)(m^2 + m\rho_{i+1}) - \pi_n$$

$$\left(1 + \frac{\rho_n}{\rho_{n-1}}\right) C^A(T) \geq C^X(T) - f_n$$

Applying these inequalities in the proof of Theorem 2 we have the result claimed.

Note here, that since $\rho_i$ is not an integer, for every $\rho_i \geq \rho_n$ task completed by $X$, it could be the case that more than $[\rho_i / \rho_n]$ groups of length $\rho_n$ are necessary to cover the execution time of $\rho_i$.  

Nonetheless, the tasks completed in each $i$-group, for $i \in [2,k]$ might be of different sizes, making the comparison of the completed load competitiveness of the above claim with the completed load competitiveness of Theorem 2 ambiguous. Another modification is therefore necessary in order to tackle this uncertainty, and we present it with algorithm $Mk$-Amortized (see pseudo-code in Alg. 4).

Algorithm description. Algorithm $Mk$-Amortized completes tasks of the same size as long as possible and change only when it is necessary. For that, we split the execution into stages of total length $ck\pi_k$, where $c \in \mathbb{N}$ is a fixed large constant. At the beginning of a stage, a set of candidate task sizes is established as $C = \left\{ i \pi_i \left\lceil \frac{Lw}{m} \right\rceil \geq c k \pi_k \right\}$. Then, the appropriate size $\pi_i^*$ is set as the minimum size in the set of candidate sizes, i.e., $i^* = \min(C)$. This is the size of tasks that the algorithm will start executing. The appropriate size is updated after every task completion, checking first whether there has been some change in the set of candidate tasks – due to new task injections. More details are given by the pseudo-code in Algorithms 4 and 8.

As a first observation, let us clearly note that following algorithm $Mk$-Amortized, only tasks of size $\pi_i^*$ are scheduled, unless there are not enough tasks to guarantee non-redundancy (when set $C = \emptyset$), in which case, a task $\pi$ at position $p \cdot m$ of the whole queue $Q$ is scheduled. To see this clearly, look at lines 6-12 in Alg. 4 and lines 4, 7 and 8 in Alg. 5. It is also important to note that parameter $i^*$ can only decrease in each stage. This is because at the beginning of the stage there are enough pending $\pi_i$-tasks for each candidate.
Algorithm 4 Mk-Amortized (for machine \( p \))

1 Parameters: \( m, \{\pi_1, \pi_2, \ldots, \pi_k\} \)

2 Upon awaking or restart

3 Repeat

4 Get all pending queues from the Repository, \( L_i \);

5 \( C \leftarrow \{i \mid \frac{|L_i|}{m^2} \geq ck \pi_k \} \);

6 While \( \{i \mid \frac{|L_i|}{m^2} \geq ck \pi_k \} = \emptyset \) Do

7 execute task \( \pi \) at position \( (p \cdot m) \mod |Q| \) in \( Q \);

8 Inform the Repository for the task completion;

9 \( C \leftarrow \{i \mid \frac{|L_i|}{m^2} \geq ck \pi_k \} \);

10 \( i^* \leftarrow \min(C) \);

11 For \( a = 1 \) to \( ck \) Do

12 \( \pi' \leftarrow \text{Schedule\_Group}(k) \);

Algorithm 5 Schedule\_Group(\( j \))

1 Parameters: \( m, j, \{\pi_1, \pi_2, \ldots, \pi_k\}, \{ L_1, L_2, \ldots, L_j \} \)

2 \( \pi \leftarrow 0; \)

3 While \( \pi \leq \pi_j - \pi_i \) Do

4 If \( j > i^* \) then

5 \( \pi' \leftarrow \text{Schedule\_Group}(j - 1); \)

6 \( \pi \leftarrow \pi + \pi'; \)

7 Else

8 execute task \( \pi_j \) at position \( p \cdot m \) in \( L_i \);

9 If task \( \pi_j \) completed successfully then

10 Inform the Repository for the task completion;

11 \( \pi \leftarrow \pi_j; \)

12 \( C \leftarrow C \cup \{i \mid \frac{|L_i|}{m^2} \geq ck \pi_k \} \);

13 \( i^* \leftarrow \min(C) \);

14 Return \( \pi \)

Task size \( i \in C \) to cover a time interval of length \( ck \pi_k \) (line 9 of Alg. \( 4 \)). Also, like algorithm k-Amortized, the modified algorithm Mk-Amortized belongs to the GroupLIS algorithms and has the property of non redundancy when enough tasks are pending, which we show in the following lemma.

Lemma 9 Algorithm Mk-Amortized never completes the same task more than once within Schedule\_Group (Alg. \( 5 \)).

Proof: Let us begin by showing that the algorithm schedules a \( \pi_j \)-task in function Schedule\_Group only when there are at least \( m^2 \) tasks in the corresponding pending queue \( L_j \). Looking at the pseudo-code, a task \( \pi_j \) is scheduled in line 8 of Alg. \( 4 \) only when the condition in line 4 does not hold, and hence \( j = i^* \). From lines 9 and 10 of Alg. \( 4 \) and lines 12 and 13 of Alg. \( 5 \) we know that \( i^* \) belongs to the set of candidate task sizes, for which every task size has at least as many tasks pending as necessary to “cover” \( ck \pi_k \) time, i.e., \( ck \) calls to the Schedule\_Group(\( j \)) function. This number of tasks is:

\[
\pi_j \left\lfloor \frac{|L_j|}{m^2} \right\rfloor \geq ck \pi_k \Rightarrow \left\lfloor \frac{|L_j|}{m^2} \right\rfloor \geq ck \lfloor \rho_k,i \rfloor \Rightarrow |L_i| \geq (ck \lfloor \rho_k,i \rfloor + 1)m^2.
\]

What is more, since \( |L_i| \geq m^2 \), Lemma \( 1 \) holds for algorithm Mk-Amortized as well (it belongs to the GroupLIS algorithms), and hence combining the two, we have the property claimed.
Theorem 3 Algorithm Mk-Amortized can reach the optimal completed-load competitiveness, theorem.

Consider now adapting the value of $c$ in the executions of algorithm Mk-Amortized; in particular, gradually increasing it to $2c$ when the total completed load of the executed tasks is big enough to guarantee the current competitive ratio close enough to $c'\gamma$ for the current value of $c$. Following this adaptation, the completed load competitiveness will get arbitrarily close to $\gamma$ after sufficiently long time, giving the following theorem.

Theorem 3 Algorithm Mk-Amortized can reach the optimal completed-load competitiveness, $\min_{1 \leq j < i \leq k} \left\{ \frac{\pi_{i,j}}{\pi_{j} \cdot \rho_{i,j}} \right\} \cdot c'$, where $c'$ is a constant that depends on parameter $c$ of the algorithm. For large enough $c$, $c'$ can be arbitrarily close to 1.

4 Speedup

Let us now look at the case in which the machines have speedup $s \geq 1$. As we have mentioned, the negative results (upper bounds) of completed-load competitiveness of any work-conserving algorithm $ALG_W$ shown for the setting of 1 machine, still hold for the case of $m$ machines. However, the positive results (lower bounds) may not. In this case, we show that for specific amounts of speedup two positive results are preserved in the multiple machine setting.

Theorem 4 Any distributed work-conserving algorithm $ALG_W$, running on a system with $m$ machines with speedup $s \geq \rho$, that guarantees non redundant executions while there are at least $m^2$ tasks pending, has a completed-load competitive ratio $C(ALG_W) \geq 1/\rho$.

Proof: We consider any distributed work-conserving algorithm $ALG_W$, running on $m$ parallel machines with speedup $s \geq \rho$. For the proof of the theorem we consider only the periods of execution during which there are at least $m^2$ pending tasks and look at the number of pending and completed tasks. During the remaining time of the executions, the completed load is bounded by the number of tasks pending (i.e., $< m^2 \pi_{\text{max}}$).

Let us then consider the execution of each machine of the system individually, and look at the number of completed tasks. In particular, observe first that for time instant $t = 0$, at the beginning of an execution of each machine $p$, $|N_p(A,0)| \geq |N_p(X,0)|$. Then consider any time instant $t > 0$ and a corresponding $t' < t$ in its execution, such that $t'$ is the latest time before $t$ that the machine has either crashed or restarted. By the definition of $t'$ there are always at least $m^2$ tasks within interval $T = (t',t]$. By induction hypothesis, at time $t'$, $|N_p(A,t')| \geq |N_p(X,t')|$.

Now let $I_T$ be the number of tasks injected during interval $T$. Since $ALG$ is a work-conserving algorithm, it is continuously scheduling and executing tasks in the interval $T$. What is more, we know that it need at most $\frac{\pi_{\text{max}}}{s} \leq \pi_{\text{min}}$ time to execute any task, since $s \geq \rho = \frac{\pi_{\text{max}}}{\pi_{\text{min}}}$. This means that $|N_p(A,t)| \geq |N_p(A,t')| + \left\lfloor \frac{t-t'}{\pi_{\text{max}}/s} \right\rfloor \geq |N_p(A,t')| + \left\lfloor \frac{t-t'}{\pi_{\text{min}}} \right\rfloor$. On the other hand, algorithm $X$ needs at least $\pi_{\text{min}}$ time to complete a task, which means that $|N_p(X,t)| \leq |N_p(X,t')| + \left\lfloor \frac{t-t'}{\pi_{\text{min}}} \right\rfloor$. This results to $|N_p(A,t)| \geq |N_p(X,t)|$.

Combining the result for all machines of the system we have that $|N(A,t)| = \sum_{i=1}^{m} |N_i(A,t)| \geq \sum_{i=1}^{m} |N_i(X,t)| = |N(X,t)|$. This leads to the desired completed-load competitiveness, $C(ALG_W) \geq 1/\rho$ since $ALG_W$ may be completing only $\pi_{\text{min}}$-tasks while $X$ completes $\pi_{\text{max}}$ ones.

Theorem 5 Any distributed work-conserving algorithm $ALG_W$ running in a system with $m$ machines and speedup $s \geq 1 + \rho$, that guarantees non redundant executions while there are at least $m^2$ tasks pending, has completed-load competitive ratio $C(ALG) \geq 1$. 

16
In [12], we studied some of the most popular algorithms in task scheduling, in the setting of one machine and analyzed their complete-load competitiveness under various ranges of speedup. Algorithm LIS becomes 1-completed-load competitive as soon as $s \geq \max\{\rho, 2\}$. However, when looking at its performance in the setting of $m$ machines (see its pseudo-code Alg. 6 in the Appendix), we realized that even in the case of 2 machines, it may not achieve 1-completed-load competitiveness.

**Theorem 6** When algorithm $m$-LIS runs in a parallel system of two machines ($m = 2$) and speedup $s = \rho = 2$, it is not 1-completed-load competitive, i.e., $C(m$-LIS) $< 1$.

5 Conclusions

In this work, we present the problem of online distributed scheduling of tasks with different computational demands on fault-prone parallel systems. We conduct worst-case analysis of deterministic work-conserving algorithms, looking at their completed-load competitiveness as the performance metric.

We show that the upper bound shown for the case of a single machine and no speedup in [11] can be achieved in our setting with $m$ machines, making the result a tight bound. Additionally, the algorithms for scheduling packets of $k$ packet lengths in one link in [19] can also be adapted to task scheduling in one machine, and then non-trivially generalized to $m$ machines. Hence, we present algorithms for the cases of two or $k$ different task sizes that achieve optimal completed load when run without speedup.

We also show that in the case of speedup, $s > 1$, the competitiveness can be improved. In particular, when speedup is $s \geq 1 + \rho$, any deterministic work-conserving algorithm ALGw may achieve optimal completed load $C(ALG) = 1$. However, we also give a negative result for algorithm $m$-LIS, the natural parallel version of the popular Longest In System scheduling policy. We show, that while with 1 machine it achieves 1-completed-load competitiveness with speedup $s \geq \max\{\rho, 2\}$, in a system of two machines running with speedup $s = \rho = 2$ its completed-load competitiveness is $C(m$-LIS) $< 1$.

There are still a few open questions though, some of which we would like to answer in future works. We believe that $m$-LIS is an important and interesting algorithm, for its popularity and fairness. As we just mentioned, the completed-load competitiveness depends on the number of machines $m$. It would be interesting to understand better the exact relation between $m$, $s$, and completed-load competitiveness in $m$-LIS. A second concrete question is whether there is a scheduling algorithm that achieves 1-completed load competitiveness with speedup $s < 1 + \rho$.

6 Acknowledgements

Supported in part by the grant TEC2014-55713-R of the Spanish Ministry of Economy and Competitiveness (MINECO), the Regional Government of Madrid (CM) grant Cloud4BigData (S2013/ICE-2894, co-funded by FSE & FEDER), the NSF of China grant 61520106005, and European Commission H2020 grants ReCred and NOTRE. Also partially supported by the FPU12/00505 grant from the Spanish Ministry of Education, Culture and Sports (MECD).

References

[1] S. Albers, A. Antoniadis, and G. Greiner. On multi-processor speed scaling with migration: extended abstract. In Proceedings of the 23rd ACM symposium on Parallelism in algorithms and architectures, SPAA ’11, pages 279–288, New York, NY, USA, 2011. ACM.

[2] S. Anand, N. Garg, and N. Megow. Meeting deadlines: How much speed suffices? In Proceedings of the 38th International Colloquium on Automata, Languages and Programming (ICALP 2011), pages 232–243, 2011.
[3] Antonio Fernández Anta, Chryssis Georgiou, Dariusz R Kowalski, and Elli Zavou. Online parallel scheduling of non-uniform tasks: trading failures for energy. *Theoretical Computer Science*, 2015.

[4] Jordi Arjona Aroca, Angelos Chatzipapas, Antonio Fernández Anta, and Vincenzo Mancuso. A measurement-based analysis of the energy consumption of data center servers. In *Proceedings of the 5th International Conference on Future Energy Systems*, e-Energy ’14, pages 63–74, New York, NY, USA, 2014. ACM.

[5] B. Awerbuch, S. Kutten, and D. Peleg. Competitive distributed job scheduling (extended abstract). In *Proceedings of the twenty-fourth annual ACM symposium on Theory of computing*, STOC ’92, pages 571–580, New York, NY, USA, 1992. ACM.

[6] Anju Bala and Inderveer Chana. Fault tolerance-challenges, techniques and implementation in cloud computing. *IJCSI International Journal of Computer Science Issues*, 9(1):1694–0814, 2012.

[7] H.L. Chan, J. Edmonds, and K. Pruhs. Speed scaling of processes with arbitrary speedup curves on a multiprocessor. In *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, SPAA ’09, pages 1–10, New York, NY, USA, 2009. ACM.

[8] Ho-Leung Chan, Joseph Wun-Tat Chan, Tak-Wah Lam, Lap-Kei Lee, Kin-Sum Mak, and Prudence WH Wong. Optimizing throughput and energy in online deadline scheduling. *ACM Transactions on Algorithms (TALG)*, 6(1):10, 2009.

[9] Jaliya Ekanayake and Geoffrey Fox. *Cloud Computing: First International Conference, CloudComp 2009 Munich, Germany, October 19–21, 2009 Revised Selected Papers*, chapter High Performance Parallel Computing with Clouds and Cloud Technologies, pages 20–38. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[10] Hesham El-Rewini, Hesham H Ali, and Ted Lewis. Task scheduling in multiprocessing systems. *Computer*, 28(12):27–37, 1995.

[11] Antonio Fernández Anta, Chryssis Georgiou, Dariusz R. Kowalski, Joerg Widmer, and Elli Zavou. Measuring the impact of adversarial errors on packet scheduling strategies. *Journal of Scheduling*, pages 1–18, 2015.

[12] Antonio Fernández Anta, Chryssis Georgiou, Dariusz R. Kowalski, and Elli Zavou. Adaptive resource management and scheduling for cloud computing. In *Adaptive Resource Management and Scheduling for Cloud Computing - Second International Workshop, ARMS-CC 2015, held in Conjunction with ACM Symposium on Principles of Distributed Computing, PODC 2015, Donostia-San Sebastián, Spain, July 20, 2015, Revised Selected Papers*, pages 1–16, 2015.

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

[14] Chryssis Georgiou and Dariusz R Kowalski. On the competitiveness of scheduling dynamically injected tasks on processes prone to crashes and restarts. *Journal of Parallel and Distributed Computing*, 2015.

[15] Anis Gharbi and Mohamed Haouari. Optimal parallel machines scheduling with availability constraints. *Discrete Applied Mathematics*, 148(1):63–87, 2005.

[16] K.S. Hong and J.Y.-T Leung. On-line scheduling of real-time tasks. In *Real-Time Systems Symposium, 1988., Proceedings., pages 244–250, 1988.

[17] Ravi Jhawar, Vincenzo Piuri, and Marco Santambrogio. Fault tolerance management in cloud computing: A system-level perspective. *Systems Journal, IEEE*, 7(2):288–297, 2013.
A Omitted Proofs

Completed-load of algorithm $\rho m$-Preamble

Proof of Lemma

Let us fix a pair of arrival and error patterns, such that executions of case (b) occur. Let us now look at the scheduling decisions and performance of each machine individually, after the defined time instant $t$. Note that in such a case, there will only be time intervals of type $T^+$ and/or $T^-$. Otherwise, the execution would be of case (a) since for every time instant $t$ there would exist a future $t' > t$ for which $|L_{\pi_{\text{min}}}(A, t')| < \rho m^2 \land |L_{\pi_{\text{max}}}(A, t')| < m^2$ would hold. We define two types of periods for the machine status: the active and the inactive periods. During an active period the machine remains alive and the queue of pending tasks does not become empty (recall that the queue of pending tasks never becomes empty in the execution we are studying). An inactive period is a non-active one. In other words, a time interval $[t_r, t_c)$ is active if it starts with time instant $t_r$ such that it is the time right after a restart of the machine. Correspondingly, it ends with time instant $t_c$ such that the machine crashes. We then focus on the active periods, with length $\lambda$, which are further categorized in the following four kinds of phases:

We safely ignore the inactive ones since the queue of pending tasks does not become empty and the algorithm $\rho m$-Preamble is work-conserving. Hence inactive periods are only while the machine is still crashed.
1. Starts with $\pi_{\text{min}}$-tasks and has length $\lambda < \overline{p}\pi_{\text{min}}$.
2. Starts with $\pi_{\text{min}}$-tasks and has length $\lambda \geq \overline{p}\pi_{\text{min}}$.
3. Starts with $\pi_{\text{max}}$-tasks and has length $\lambda < \pi_{\text{max}}$.
4. Starts with $\pi_{\text{max}}$-tasks and has length $\lambda \geq \pi_{\text{max}}$.

Let us look at the $i^{\text{th}}$ period after time $t$ in the execution of $\rho\text{-Preamble}$. Let us also denote by $a_i$ the number of completed $\pi_{\text{min}}$-tasks, apart from the $\overline{p}$ preamble, by $b_i$ the number of completed $\pi_{\text{max}}$-tasks and by $c_i$ the number of completed $\pi_{\text{min}}$-tasks in the preamble. For the execution of $X$ we denote by $a_i^*$ the total number of completed $\pi_{\text{min}}$-tasks and by $b_i^*$ the total number of completed $\pi_{\text{max}}$-tasks. Let also $C^A(i_j)$ and $C^X(i_j)$ denote the total completed load within a phase $i$ of type $j$ by $\rho\text{-Preamble}$ and $X$ respectively. Analyzing the four types of active periods, we make the following observations.

For phases of type 1, $\rho\text{-Preamble}$ is not able to complete the $\overline{p}$ $\pi_{\text{min}}$ tasks of the preamble, while $X$ is only able to complete at most as much load, so $\sum_{j>i} C^X(i_j) \leq \sum_{j>i} C^A(i_j)$.

For phases of type 2, the total completed load by $X$ minus the completed load by $\rho\text{-Preamble}$ is at most $\pi_{\text{max}}$ (i.e., $\sum_{j>i} (C^X(i_j) - C^A(i_j)) < \pi_{\text{max}}$). Therefore,

$$\sum_{j>i} C^A(i_j) \geq \frac{\overline{p}\pi_{\text{min}}}{\pi_{\text{max}} + \overline{p}\pi_{\text{min}}} \sum_{j>i} C^X(i_j).$$

(Observe that $\frac{\overline{p}\pi_{\text{min}}}{\pi_{\text{max}} + \overline{p}\pi_{\text{min}}} \leq 1/2$.)

The same holds for phases of type 4 and hence $\sum_{j>i} C^X(i_4) \leq 2 \sum_{j>i} C^A(i_4)$.

In phases of type 3, $\rho\text{-Preamble}$ is not able complete any task and hence $\sum_{j>i} C^A(i_3) = 0$, whereas $X$ might complete up to $(\lceil \rho \rceil - 1)\pi_{\text{min}}$ tasks. There are two cases of executions to be considered then:

**Case 1:** The number of phases of type 3 is finite.

In this case, there is a phase $i^*$ such that $\forall i > i^*$ phase $i$ is not of type 3. Then,

$$C_1(A) = \frac{\sum_{j \leq i^*} C^A(j) + \sum_{j > i^*} C^A(j)}{\sum_{j \leq i^*} C^X(j) + \sum_{j > i^*} C^X(j)}$$

(4)

Observe that the total progress completed by the end of phase $i^*$ by both algorithms is bounded. So for simplicity, we overload notations $A$ and $X$ and define $\sum_{j \leq i^*} C^A(j) = A$ and $\sum_{j \leq i^*} C^X(j) = X$. Therefore,

$$C_1(A) = \frac{A + \sum_{j > i^*} C^A(j)}{X + \sum_{j > i^*} C^X(j)} \geq \frac{A + \frac{\overline{p}\pi_{\text{min}}}{\pi_{\text{max}} + \overline{p}\pi_{\text{min}}} \sum_{j > i^*} C^X(j)}{X + \sum_{j > i^*} C^X(j)}.$$

Hence, the completed load competitiveness of $\rho\text{-Preamble}$ at the end of each phase can be computed as $C(\rho\text{-Preamble}) = \lim_{t \to \infty} C_1(A)$, i.e.,

$$C(\rho\text{-Preamble}) = \lim_{j \to \infty} \left( \frac{A + \frac{\overline{p}\pi_{\text{min}}}{\pi_{\text{max}} + \overline{p}\pi_{\text{min}}} \sum_{j > i^*} C^X(j)}{X + \sum_{j > i^*} C^X(j)} \right)$$

$$= \lim_{j \to \infty} \left( \frac{\overline{p}\pi_{\text{min}}}{\pi_{\text{max}} + \overline{p}\pi_{\text{min}}} + \frac{(\pi_{\text{max}} + \overline{p}\pi_{\text{min}})A - (\overline{p}\pi_{\text{min}})X}{(\pi_{\text{max}} + \overline{p}\pi_{\text{min}})(X + \sum_{j > i^*} C^X(j))} \right)$$

$$= \frac{\overline{p}\pi_{\text{min}}}{\pi_{\text{max}} + \overline{p}\pi_{\text{min}}} = \frac{\overline{p}}{\rho + \overline{p}}.$$
It is important to note that the assumption $\lim_{t \to \infty} C^X(t) = \infty$ is used, which corresponds to the expression $\lim_{j \to \infty} \sum_{j \geq i^*} C^X(j)$ in the above equality.

The above analysis shows the completed-load competitiveness at the end of each phase. However, we have to guarantee that the lower bound holds at all times within the phases. For this, consider any time instant $t$ of phase $i > i^*$. At that instant $C_i(t) = \sum_{j \in (i^*, i-1]} C^A(j) + A_i + \sum_{j \in (i^*, i-1]} C^X(j) + X_t$, where $A_i$ and $X_t$ represent the load completed by $\rho$-m-Preamble and $X$ within phase $i$ up to time $t$. Using the above proof, and the fact that for phases of type 1, 2 and 4 we have

$$\sum_{\forall i} (C^A(i_1) + C^A(i_2) + C^A(i_4)) \geq \frac{\lim_{t \to \infty} \rho}{\pi_{\max} + \pi_{\min} \rho} \cdot \sum_{\forall i} (C^X(i_1) + C^X(i_2) + C^X(i_4)),$$

we know that $A_i \geq \frac{\lim_{t \to \infty} \rho}{\pi_{\max} + \pi_{\min} \rho} \cdot X_t$ as well. Hence,

$$C_i(t) \geq \frac{\rho \pi_{\min}}{\pi_{\max} + \pi_{\min}} \sum_{j \in (i^*, i-1]} C^X(j) + \frac{\rho \pi_{\min}}{\rho + \rho} X_t \sum_{j \in (i^*, i-1]} C^X(j) + X_t = \frac{\rho \pi_{\min}}{\pi_{\max} + \rho \pi_{\min}} = \frac{\rho}{\rho + \rho}.$$

**Case 2:** The number of phases of type 3 is infinite.

In this case we must show that the number of $\pi_{\min}$ and $\pi_{\max}$-tasks completed are bounded for both $\rho$-m-Preamble and $X$.

**Claim A.1** Consider the time instant $t$ at the beginning of a phase $j$ of type 3. Then the number of $\pi_{\min}$-tasks completed by $X$ by time $t$ is no more than the number of $\pi_{\min}$-tasks completed by $\rho$-m-Preamble, plus $\rho - 1$, i.e., $\sum_{i < j} a_i^* \leq \sum_{i < j} (a_i + c_i) + (\rho - 1)$.

**Proof:** Consider the beginning of phase $j$ of type 3. We know that at that time instant algorithm $\rho$-m-Preamble has at most $(\rho - 1)$ $\pi_{\min}$-tasks pending. Recall that a machine following algorithm $\rho$-m-Preamble, after restarting it first completes a preamble of $\rho \pi_{\min}$ tasks, before executing any $\pi_{\max}$ ones. By the definition of type 3, it may only occur if there are not enough $\pi_{\min}$-tasks pending at time instant $t$. Hence, the amount of $\pi_{\min}$-tasks completed by $X$ by the beginning of phase $j$ is no more than the ones completed by algorithm $\rho$-m-Preamble (including the ones in preambles) plus $\rho - 1$. \qed

**Claim A.2** Considering all types of phases and the number of $\pi_{\max}$-tasks completed, it holds that $\sum_{i \leq j} b_i^* \leq \sum_{i \leq j} b_i + \sum_{i \leq j} \frac{c_i}{\rho} + 2$, for every phase $j$.

**Proof:** To prove this, we use induction on phase $j$.

**Base Case:** For $j = 0$ the claim is trivial.

**Induction Hypothesis:** It holds that

$$\sum_{i \leq j} b_i^* \leq \sum_{i \leq j} b_i + \sum_{i \leq j} \frac{c_i}{\rho} + 2.$$

**Induction Step:** We need to prove that the relationship holds up to the end of phase $j$. Consider first that during phase $j$ there is a time when $\rho$-m-Preamble has no $\pi_{\max}$-tasks pending, and let $t$ be the latest such time in the phase. We define $b^*(t)$ and $b(t)$ being the number of $\pi_{\max}$-task completed up to time $t$ by algorithm $X$ and $\rho$-m-Preamble respectively. We know that $b^*(t) \leq b(t)$. We also define $x_j^*(t)$ and $x_j(t)$ to be the number of $\pi_{\max}$-tasks scheduled by $X$ and $\rho$-m-Preamble respectively after time instant $t$ and until the
end of the phase $j$. We claim that $x^*_j(t) \leq x_j(t) + 2$. From our definitions, at time $t$ algorithm $\rho m$-Preamble is executing a $\pi_{\min}$-task. Since it is the last instant that it has no $\pi_{\max}$-task pending, the worst case is to be at the beginning of the preamble (by inspection of the 4 types of phases). Then, if the phase ends at time $t'$, period $I = [t, t')$ is such that $|I| < \pi_{\min} + (x_j(t) + 1)\pi_{\max} \leq (x_j(t) + 2)\pi_{\max}$. (The $+1\pi_{\max}$-task is because of the machine crash before completing the last $\pi_{\max}$-task scheduled in the phase.) Observe that $X$ could be executing a $\pi_{\max}$-task at time $t$, completed at some point in $[t, t + \pi_{\max}]$ and accounted for in $x^*_j(t)$. Therefore,

$$\sum_{i \leq j} b^*_i = b^*(t) + x^*_j(t) \leq b(t) + x_j(t) + 2 = \sum_{i \leq j} b_i + 2.$$  

Now consider the case where at all times of phase $j$ there are $\pi_{\max}$-tasks pending for $\rho m$-Preamble. By inspection of the 4 types of phases, the worst case is when $j$ is of type 2. After completing the preamble of $p\pi_{\min}$ tasks, the algorithm schedules $\pi_{\max}$-tasks until the machine crashes again interrupting the last one scheduled. On the same time, $X$ is able to complete at most $\frac{\lambda_j}{\pi_{\max}} \leq b_j + 1\pi_{\max}$-tasks, where $\lambda_j$ is the length of the phase. Hence, in all types of phases $b^*_j \leq \frac{\lambda_j}{\pi_{\max}} + b_j$ and by induction, the claim follows;  

$$\sum_{i \leq j} b^*_i \leq \sum_{i \leq j} \frac{\lambda_i}{\pi_{\max}} + \sum_{i \leq j} b_i + 2.$$  

Combining the two claims above, the completed load competitiveness ratio of case 2 is as follows:

$$C_2(A) = \frac{\sum_{i \leq j} C^A(i)}{\sum_{i \leq j} C^X(j)} = \frac{\sum_{i \leq j} [(a_i + c_i)\pi_{\min} + b_i\pi_{\max}]}{\sum_{i \leq j} [a^*_i\pi_{\min} + b^*_i\pi_{\max}]} \geq \frac{\sum_{i \leq j} [(a_i + c_i)\pi_{\min} + b_i\pi_{\max}]}{\sum_{i \leq j} [(a_i + c_i)\pi_{\min} + b_i\pi_{\max}]} + \frac{3\pi_{\max}}{\pi_{\max}} \geq \frac{\sum_{i \leq j} [(a_i + c_i)\pi_{\min} + b_i\pi_{\max}]}{\sum_{i \leq j} [(a_i + c_i)\pi_{\min} + b_i\pi_{\max}]} - \frac{3\pi_{\max}}{\pi_{\max}} \geq \frac{3\pi_{\max}}{2\pi_{\max}} \geq \frac{1}{2} - \frac{\pi_{\max}}{2\pi_{\max}}.$$  

Note that, due to the parameters $a_i, b_i$ and $c_i$, the second ratio tends to zero (the denominator tends to infinity) and hence the completed load competitive ratio tends to $C(\rho m$-Preamble) $= \lim_{t \to \infty} C_2(A) \geq \frac{1}{2}$. 

Combining now the results from the two cases concerning the number of phases of type 3, since $\frac{\pi_{\max}}{\rho + \rho} \leq \frac{1}{2}$, the completed load of algorithm $\rho m$-Preamble is at least $\frac{\pi_{\max}}{\rho + \rho}$ as claimed.  

**Completed-load of Algorithm Amortized**

**Proof of Lemma** Let us divide the $n$-busy interval $T = [t, t']$ of $p$ into two intervals; the first being from the beginning, $t$, to a time instant $t^* \geq t$ such that the first restart happens in the interval, and the second being the remaining of the interval, from $t^*$ to $t'$. In other words, interval $T_1 = [t, t^*]$ and $T_2 = [t^*, t']$.

Looking first at interval $T_2$, it starts by a restart and then, either includes more crashes and restarts or not, and never schedules tasks of size more than $\pi_n$. Hence, at time $t^*$ the machine starts executing a
new task with both k-Amortized and X. Also, since \( p \) is busy at all times of the interval, for every \( \pi_i \)-task completed by \( X \) in \( T_2 \) – say in interval \( T_1 = [t_1, t_2] \) in \( T_2 \) where \( t_2 = t_1 + \pi_i \) and \( i \geq n \) – the machine is able to complete \( \rho_{i,n} \) n-groups in \( T_1 \) (each of size \( \pi_n \)). These groups correspond to executions of the recursive function \( \text{Schedule\_Group}(n) \). Hence, we can assign each n-group to the task completed by \( X \) at the moment when the last task in the n-group is completed by k-Amortized, which gives inequality \( C_p^A(T_2) \geq C_p^X(T_2, \geq \pi_n) \).

Looking now at interval \( T_1 \), we must consider the following cases for the execution of \( p \):

1. At time instant \( t \) there was a restart (\( t^* = t \)) and hence the machine started executing a new task, with both \( X \) or k-Amortized. In this case the analysis of interval \( T_2 \) will hold.
2. At time instant \( t \) it is already executing a task \( \tau \) with \( X \), scheduled before \( t \) and then, it either a) gets interrupted by the crash at time \( t^* \), or b) it completes it within the interval \( T_1 \). In the first case, \( X \) is not able to complete any task in \( T_1 \) while k-Amortized may complete up to \( |T_1| \), for which it is trivial that \( C_p^A(T_1) \geq C_p^X(T_1) \) holds. In the latter, task \( \tau \) will be of maximum size \( \pi_k \). Then for the rest of the interval, the same analysis as for \( T_2 \) holds, for every \( \pi_i \)-task fully contained in the interval and completed by \( X \), where \( i \geq n \). Hence, \( C_p^A(T_1) \geq C_p^X(T_1, \geq \pi_n) - \pi_k \).

From the two intervals, we have the claim of the lemma, \( C_p^A(T) \geq C_p^X(T, \geq \pi_n) - \pi_k \).

**Proof of Lemma 7.** The idea of the proof for this lemma, is that tasks completed by algorithm k-Amortized are associated to tasks completed by \( X \) in such a way, that: (a) each task completed by k-Amortized corresponds to at most twice the size of tasks completed by \( X \) and (b) each task completed by \( X \) is associated to tasks of the same aggregate size completed by k-Amortized. This amortization follows these two rules:

1. the \( r \)-th task of size \( \pi_i \) completed by algorithm k-Amortized within \( T \), for \( i < n \), is associated to the \( r \)-th task of size \( \pi_i \) completed by \( X \) within \( T \) (if completed).
2. the completion of a task \( \tau \) of size \( \pi_i \geq \pi_n \) by \( X \), corresponds to \( \pi_i/\pi_n \) n-groups completed by k-Amortized, such that the execution of the last task of each of the groups is finished during the execution of task \( \tau \).

First, looking at rule #1 and interval \([t_1, t]\), the following two equations hold for the pending tasks at the end of the interval:

\[
P^A(t, < \pi_n) = P^A(t_1, < \pi_n) + I([t_1, t], < \pi_n) - C_p^A([t_1, t], < \pi_n),
\]

\[
P^X(t, < \pi_n) = P^X(t_1, < \pi_n) + I([t_1, t], < \pi_n) - C_p^X([t_1, t], < \pi_n),
\]

where \( I([t_1, t], < \pi_n) \) is the set of tasks smaller than \( \pi_n \) that were injected during the interval, up to time \( t \). Since they are the same for both algorithms, from the above equations we have:

\[
P^A(t, < \pi_n) - P^A(t_1, < \pi_n) + C_p^A([t_1, t], < \pi_n) = P^X(t, < \pi_n) - P^X(t_1, < \pi_n) + C_p^X([t_1, t], < \pi_n)
\]

which leads to the completed load of k-Amortized containing only small tasks, \( < \pi_n \), being bounded as:

\[
C_p^A([t_1, t], < \pi_n) \geq C_p^X([t_1, t], < \pi_n) + P^X(t, < \pi_n) - \sum_{i=1}^{n-1} (\pi_n + \pi_i)(m^2 + m\rho_{i+1}).
\]

To see why the inequality holds, look first at the pseudo-code of the algorithm, more precisely line 6 of algorithm \( X \) a task of size \( \pi_n \) is scheduled at time \( t \) only in the case when the total size of smaller pending tasks is \( P^A(t, < \pi_n) \leq \sum_{i=1}^{n-1} (\pi_n + \pi_i)(m^2 + m\rho_{i+1}) \). Recall also condition (3) of the n-busy interval of the machine; i.e., \( P^A(t_1, \pi_i) \geq P^X(t_1, \pi_i), \forall i \in [1, n] \), which also means that \( P^A(t_1, \leq \pi_n) \geq P^X(t_1, \leq \pi_n) \). Combining these properties, the inequality follows.

Now, looking at rule #2, for any task \( \tau \) of size \( \pi_i \geq \pi_n \) completed by \( X \), we have already shown in Lemma 8 that

\[
C_p^A([t_1, t]) \geq C_p^X([t_1, t], \geq \pi_n) - \pi_k.
\]
Combining the two equations, 5 and 6, the claim follows: \(2C_p^A([t_1, t]) \geq C_p^X([t_1, t]) - \sum_{i=1}^{n-1}(\pi_n + \pi_i)(m^2 + m\rho_{i+1}) - \pi_k\).

**Proof of Lemma 8** We prove this lemma by induction on \(n\).

**Base case.** For \(n = 1\), the result is immediate from Lemma 6. More precisely, since \(C_p^X(T, \geq \pi_1) = C_p^X(T)\), then \(2C_p^A(T) \geq C_p^X(T) - \pi_k\) holds directly.

**Induction Hypothesis.** We assume that the result holds for some \(n < k\); i.e., \(2C_p^A(T) \geq C_p^X(T) - f_n\).

**Inductive Step.** We show that the result still holds for \(n + 1\). For this, we split the \((n + 1)\)-busy interval \(T\) in three sub-intervals:

- **\(T_1\)** is the interval from the beginning of \(T\) to time instant \(t\) at which k-Amortized starts executing an \(\pi_{n+1}\)-task for the last time during \(T\).
- **\(T_2\)** is the interval from \(t\) to \(t' \in T\) s.t. either k-Amortized completes the \(\pi_{n+1}\)-task, or it gives up scheduling tasks of size \(\pi_{n+1}\) at \(t'\) since it now has enough smaller tasks pending to cover the \(\pi_{n+1}\)-time and there was a crash and restart of the machine.
- **\(T_3\)** from time instant \(t'\) to the end of \(T\).

For sub-interval \(T_1\) we know that Lemma 7 holds, hence

\[
2C_p^A(T_1) \geq C_p^X(T_1) + P^X(t, < \pi_{n+1}) - \sum_{i=1}^{n}(\pi_n + \pi_i)(m^2 + m\rho_{i+1}) - \pi_k. \tag{7}
\]

Let us now consider an offline algorithm \(X',\) which acts as \(X\) during \(T\), except the fact that it starts sub-interval \(T_2\) only with tasks with length at least \(\pi_{n+1}\) (no smaller ones), and stays idle whenever \(X\) executes a task that was pending in its queue at time instant \(t\) but not in the queue of \(X'\).

Note here, that algorithm k-Amortized finishes the last attempt to complete a \(\pi_{n+1}\)-task, no longer than \(\pi_{n+1}\) time after time instant \(t\), in other words, \(|T_2| \leq \pi_{n+1}\). Hence, \(C_p^X(T_2) \leq \pi_{n+1} + \pi_k\), where \(\pi_k\) comes from the possibility of \(X'\) scheduling a task before \(T_2\) and completing it within \(T_2\). Hence,

\[
2C_p^A(T_2) \geq 0 \geq C_p^X(T_2) - \pi_{n+1} - \pi_k. \tag{8}
\]

At the beginning of sub-interval \(T_3\), we have that \(P^A(t', \pi_i) \geq P^X(t', \pi_i)\) for each \(i \leq n\), since k-Amortized only attempted the execution of a \(\pi_{n+1}\) during \(T_2\) and \(X'\) starts the \(T_2\) without any tasks smaller than \(\pi_{n+1}\). This means that the inductive hypothesis holds for sub-interval \(T_3\) for the largest task \(\pi_n\) and offline algorithm \(X'\) instead of \(X\):

\[
2C_p^A(T_3) \geq C_p^X(T_3) - f_n. \tag{9}
\]

Now observe that at time instant \(t\), at the beginning of interval \(T_2\), algorithm \(X\) has \(P^X(t, < \pi_{n+1})\) more tasks pending than \(X'\). Hence, by the end of interval \(T_3\) the following will hold:

\[
C_p^X(T_2 \cup T_3) \geq C_p^X(T_2 \cup T_3) - P^X(t, < \pi_{n+1}). \tag{10}
\]

Putting equations 7 to 10 together to calculate the completed load of the total interval \(T\), we have

\[
2C_p^A(T) \geq C_p^X(T_1) + C_p^X(T_2 \cup T_3) + P^X(t, < \pi_{n+1}) - \sum_{i=1}^{n}(\pi_n + \pi_i)(m^2 + m\rho_{i+1}) - \pi_{n+1} - 2\pi_k + f_n
\]

\[
\geq C_p^X(T_1) + C_p^X(T_2 \cup T_3) - \sum_{i=1}^{n}(\pi_n + \pi_i)(m^2 + m\rho_{i+1}) + \pi_{n+1} - 2\pi_k - f_n
\]

\[
\geq C_p^X(T) - \sum_{i=1}^{n}(\pi_n + \pi_i)(m^2 + m\rho_{i+1}) + \pi_{n+1} - 2\pi_k - f_n
\]

\[
\geq C_p^X(T) - f_{n+1}
\]

which completes the induction step and thus the proof of the lemma. □
Completed-load of Algorithm MAmortized

The following definition and Claim are necessary for the proof of Lemma 10.

**Definition 7** An execution of Schedule_Group(k) is considered to be uniform, if the algorithm completes tasks of a single fixed size \( \pi_i \) only, during the executions of the current Schedule_Group(k) as well as the previous one.

**Claim A.3** There are at least \( ck - 2k \) uniform calls of Schedule_Group(k) in a stage of Mk-Amortized.

**Proof:** As already mentioned, the value of \( t^* \) can only decrease during a stage. Since there are up to \( k \) task sizes, this can happen up to \( k - 1 \) times. However, for a Schedule_Group(k) execution to be uniform, its previous execution must also be uniform. Hence, there can be at least \( ck - 2k \) calls of Schedule_Group(k) in a stage that are uniform. \( \square \)

**Proof of Lemma 10:** Let us start by assuming that all the executions of Schedule_Group in Mk-Amortized are uniform. Every task \( \pi_i \geq \pi_n \) completed by X, will correspond to a group of \( \lfloor \pi_i / \pi_n \rfloor \) tasks of total size \( \pi_i \) completed by algorithm Mk-Amortized. In particular, to the group of tasks that completed their execution successfully during the execution of the task \( \pi_i \) by X.

Now let \( \gamma = \min_{i<j<i\leq k} \{ \frac{p_{i,j}}{p_{i,j}+p_{i,j}} \} \). Let us also define \( \delta_{i,j} = \frac{p_{i,j}}{p_{i,j}+p_{i,j}} \) and \( \delta = \max\{\delta_{i,j}\} \). This means that \( \gamma = 1/(1+\delta) \). The assignment of tasks completed by Mk-Amortized, to each \( \pi_i \)-task completed by X, where \( i \geq n \), makes the inequalities from Lemmas 6, 7 and 8 as follows:

\[
\delta \cdot C^X(T) \geq C^X(T, \geq \pi_n) - \pi_k
\]

\[
(1 + \delta) \cdot C^X([t_1, t]) \geq C^X([t_1, t]) + P^X(t, < \pi_n) - \sum_{i=1}^{n}(\pi_{n+1} + \pi_i)(m^2 + m\rho_{i+1}) - \pi_n
\]

\[
(1 + \delta) \cdot C^X(T) \geq C^X(T) - f_n
\]

Then, if we apply the above inequalities in the proof of Theorem 2, we obtain the claimed result, without the \( c' \) factor, since \( \frac{1}{1+\delta} = \min_{1\leq j<i\leq k} \{ \frac{p_{i,j}}{p_{i,j}+p_{i,j}} \} \).

Nonetheless, the above case only covers the uniform executions. Let us now consider the cases where there is some execution of Schedule_Group that is not uniform. By Claim A.3, at most a fraction of calls to Schedule_Group(k) are not uniform; that is, \( 2/c \) of them. From Claim 3.1, even without the uniform executions we have the completed load competitivity at least \( \eta = \min_{s \in [2,k]} \{ \frac{\rho_s - 1}{2\rho_s - 1} \} \).

Combining the two cases, we separate the uniform and the non-uniform executions, and denote the corresponding completed load of X by \( C_1^X(T) \) and \( C_2^X(T) \) respectively. We therefore have the following relationships between the two algorithms:

\[
\left(1 - \frac{2}{c}\right) \cdot C^A(T) \geq \eta \cdot C_1^X(T) - c_1
\]

\[
\frac{2}{c} \cdot C^A(T) \geq \eta \cdot C_2^X(T) - c_2
\]

for constants \( c_1 \) and \( c_2 \) that depend on the task sizes. This means that

\[
\left(1 - \frac{2}{c} + \frac{2\eta}{c}\right) \cdot C^A(T) \geq \eta \cdot \left(C_1^X(T) + C_2^X(T)\right) - c_1 - c_2
\]

which leads to the desired completed load competitiveness:

\[
C^A(T) \geq \frac{1}{1 + 2/(c\eta)} \cdot \eta \cdot C^X(T) - c_1 - c_2
\]

where we can define the \( c' \) of the lemma equal to \( \frac{1}{1 + 2/(c\eta)} \) and choose such a \( c \), large enough, to make \( c' \) arbitrarily close to 1. This completed the proof of the lemma. \( \square \)
Completed-load in the case of speedup

**Proof of Theorem 5** We consider any distributed work-conserving algorithm ALG, running on m parallel machines with speedup s ≥ 1 + ρ. For the proof of the theorem let us consider only the periods of execution during which there are at least m² pending tasks. During the remaining time of the executions, the completed load is bounded by the number of tasks pending (i.e., < m²πmax).

Consider then, the execution of machine p and the corresponding execution of offline algorithm X in the same machine. We will be looking at their completed load by machine p at different time instances. Let us look at any time t and define time instant t' < t to be the latest time before t at which one of the following two events happens: (1) and active period starts (t' is a restart point of p), or (2) algorithm X has successfully completed a task.

It is then trivial that Cπ₀(t)(ALG, A, E) ≥ Cπ₀(t′)(X, A, E) holds at the beginning of the executions. Now, assuming that Cπ₀(t)(ALG, A, E) ≥ Cπ₀(t′)(X, A, E) holds at time t′, we prove by induction that Cπ₀(t)(ALG, A, E) ≥ Cπ₀(t′)(X, A, E) still holds at time t. This also means that the tasks successfully completed by machine p in the execution of ALG by time t have at least the same total size as the ones completed by X.

Looking at the interval T = (t′, t], we have to consider the following two cases:

**Case 1:** X is not able to complete any task in the interval T. This means that Cπ₀(T) = Cπ₀(t′) in ALG. For ALG it holds that Cπ₀(t') ≥ Cπ₀(t)(ALG, A, E), even if it is not able to complete any tasks in T. Therefore, Cπ₀(ALG, A, E) ≥ Cπ₀(t′)(X, A, E).

**Case 2:** X completes a task in the interval T. Note that, due to the definition of t′, there can only be one task completed by X within T, and it must be completed exactly at time instant t. Then, interval T has length equal to a task π ∈ [πmin, πmax]; the size of the task completed by X. In T algorithm ALG executes tasks continuously, whose aggregate size is at least πs − πmax. Then, the completed load of machine p with the two algorithms at t satisfies Cπ₀(X, A, E) = Cπ₀(t′)(X, A, E) + π and Cπ₀(ALG, A, E) ≥ Cπ₀(t′)(ALG, A, E) + (πs − πmax). Observe that the fact that s ≥ 1 + ρ implies that πs − πmax ≥ π. Hence, Cπ₀(ALG, A, E) ≥ Cπ₀(t′)(X, A, E).

The above analysis shows that the completed-load competitiveness at any time of the execution of each machine is 1. Therefore, taking the sum of completed loads of all machines, gives the claimed result; the completed-load competitiveness ratio of any work-conserving distributed algorithm that guarantees non redundant executions while there are at least m² tasks pending, is C(ALG) ≥ 1, when s ≥ 1 + ρ.

Let us present here the pseudo-code of algorithm m-LIS.

**Algorithm 6 m-LIS** (for machine p)

1. Parameters: m, πmin, πmax
2. Upon awakening or restart
3. Repeat
4. Get sorted queue Q from the Repository;
5. If |Q| ≥ m² then
6. Schedule task π at position p · m in Q;
7. else
8. Schedule task π at position (p · m) mod |Q|;
9. Inform Repository of completion of task π;

**Proof of Theorem 6 (Sketch):** We look at the case when m-LIS runs in a parallel system of two machines (m = 2) and speedup s = ρ = 2. Let us fix an adversarial strategy, consisting of task arrival and machine error patterns A and E, that work as follows:

We define δ = ρ1/3 ≈ 1.15, and use only tasks of sizes x · πmin, for x ∈ {1, δ, δ², δ³, δ⁴, δ⁵}. For simplicity, in the rest of the proof we remove the factor πmin, which only introduces a scaling factor.

The arrival pattern A is the following sequence of task sizes that is repeated over and over:

\[1, δ², δ³, δ, δ³, δ⁴, δ⁵, \ldots\] (11)
Let the arrival of tasks be fast enough so that whenever algorithms $m$-LIS or $X$ are supposed to schedule a task in the description below such a task is in the repository.

The execution then behaves as follows: We divide the execution in *epochs*, so that in one epoch $m$-LIS executes the tasks that are in positions 1 to 8 in the repository sorted by arrival time, as shown above. As defined by $m$-LIS, processor 1 always schedules the task in position 1, while processor 2 schedule the task in position 3. In each epoch there are 8 *phases* as follows:

1. In the first phase, processor 1 is crashed, while processor 2 restarts, is active for time $\delta^3$, and the crashes again. In this phase, $X$ schedules and completes a task of length $\delta^3$. On its hand, $m$-LIS schedules and completes a task of length $\delta^2$, and schedules a task of length $\delta^4$ that is interrupted, since $\frac{\delta^2+\delta^4}{2} > \delta^3$.
2. In the second phase, processor 2 stays crashed. Processor 1 is active for time $\delta$. In this phase $X$ schedules and completes a task of length $\delta$. $m$-LIS schedules and completes a task of length 1, and schedules a task of length $\delta^2$ that is interrupted.
3. In the third phase, processor 2 remains crashed. The tasks in positions 1 and 2 at the start of the task have lengths $\delta^2$ and $\delta^4$. In this phase, processor 1 is active for $\delta^3$ time. As in the first phase, $X$ completes a task of length $\delta^3$, while $m$-LIS only completes a task of length $\delta^2$.
4. In the fourth phase, processor 2 keeps being crashed. The tasks in positions 1 and 2 at the start of the task have lengths $\delta^4$ and $\delta$. In this phase, processor 1 is active for $\delta^2$ time. Hence, $X$ completes a task of length $\delta^2$ while $m$-LIS only completes a task of length $\delta^4$.

These first four phases complete the execution of the 4 first tasks in the sequence 11 above. The next three phases of the epoch are similar to phases 1 to 3, but all task lengths have an additional factor $\delta$. In the final eighth phase, only processor 1 is active, for $\delta^2$ time, $X$ completes a task of length $\delta^2$, and $m$-LIS completes a task of length $\delta^5$ (the next task in arrival order is the task of length 1 that starts a new sequence of 8 tasks like sequence 11 above).

The total length of the tasks completed by $X$ in the first epoch is then $C(X) = 2\delta^4 + 2\delta^3 + 3\delta^2 + \delta \approx 11.62$, while $m$LIS has completed $C(m$-LIS) = $\delta^5 + \delta^4 + 2\delta^3 + 2\delta^2 + \delta + 1 \approx 11.56$. Exactly the same behavior is repeated in every epoch. Hence, $m$LIS is not 1-competitive with $s = \rho = m = 2$.  

\[\square\]