Performing Dynamically Injected Tasks on Processes Prone to Crashes and Restarts

Chryssis Georgiou*
Department of Computer Science,
University of Cyprus
chryssis@cs.ucy.ac.cy

Dariusz R. Kowalski†
Department of Computer Science,
University of Liverpool
D.Kowalski@liverpool.ac.uk

September 1, 2011

Abstract

To identify the tradeoffs between efficiency and fault-tolerance in dynamic cooperative computing, we initiate the study of a task performing problem under dynamic processes' crashes/restarts and task injections. The system consists of n message-passing processes which, subject to dynamic crashes and restarts, cooperate in performing independent tasks that are continuously and dynamically injected to the system. The task specifications are not known a priori to the processes. This problem abstracts todays Internet-based computations, such as Grid computing and cloud services, where tasks are generated dynamically and different tasks may be known to different processes. We measure performance in terms of the number of pending tasks, and as such it can be directly compared with the optimum number obtained under the same crash-restart-injection pattern by the best off-line algorithm. Hence, we view the problem as an online problem and we pursue competitive analysis. We propose several deterministic algorithmic solutions to the considered problem under different information models and correctness criteria, and we argue that their performance is close to the best possible offline solutions. We also prove negative results that open interesting research directions.

Keywords: Performing tasks; Dynamic task injection; Crashes and restarts; Competitive analysis; Distributed Algorithms.

Full version of DISC 2011 paper.

*The work of this author is supported in part from research funds of the University of Cyprus.

†The work of this author is supported by the Engineering and Physical Sciences Research Council [grant numbers EP/G023018/1, EP/H018816/1].

1 Introduction

Motivation. One of the fundamental problems in distributed computing is to have a collection of processes to collaborate in performing large sets of tasks. For such distributed collaboration to be effective it must be designed to cope with dynamic perturbations that occur in the computation medium (e.g., processes or communication failures). For this purpose, a vast amount of research has been dedicated over the last two decades in developing fault-tolerant algorithmic solutions and frameworks for various versions of such cooperation problems (e.g., [11, 15, 20, 21, 26, 28]) and in deploying distributed collaborative systems and applications (e.g., [3, 14, 23, 25]).

In order to identify the tradeoffs between efficiency and fault-tolerance in distributed cooperative computing, much research was devoted in studying the abstract problem of using n processes to cooperatively perform m independent tasks in the presence of failures (see for example [13, 20, 22]). In this problem, known as *Do-All*, the number of tasks m is assumed to be fixed and known a priori to all processes. Although there are several applications in which the knowledge of tasks can be known a priori, in todays typical Internet-based computations, such as Grid computing (e.g., [14]), Cloud services (e.g., [3]), and master-worker computing (e.g., [23, 25]), tasks are generated dynamically and different tasks may be known to different processes. As such computations are becoming the norm (and not the exception) there is a corresponding need to develop efficient and fault-tolerant algorithmic solutions that would also be able to cope with dynamic tasks injections.

Our Contributions. In this work, in an attempt to identify the tradeoffs between efficiency and fault-tolerance in dynamic cooperative computing, we initiate the study of a task performing problem in which n message-passing processes, subject to dynamic crashes and restarts, cooperate in performing independent tasks that are continuously and dynamically injected to the system. The computation is broken into synchronous rounds, in which each process is injected tasks, receives messages sent to it in the prior round, performs local computations (including performing at most one task), and sends messages (if any). Unless otherwise stated, we assume that tasks are of unit-length, that is, it takes one round for a process to perform a task. An execution of an algorithm is specified under a crash-restart-injection pattern. Then, the efficiency of an algorithm is measured in terms of the *maximum number of pending tasks* at the beginning of a round of an execution, taken over all rounds and all executions. This enables us to view the problem as an online problem and pursue competitive analysis [29], that is, compare the efficiency of a given algorithm with the efficiency of the best offline algorithm that knows a priori the crash-restart-injection patterns.

Task performance guarantees: We consider two versions of the problem with respect to the task performance guarantees required by algorithmic solutions. The first one, which constitutes the basic *correctness* property, requires that no task is lost, that is, a task is either performed or the information of the task remains in the system. The second and stronger property, which we call *fairness*, requires that all tasks injected in the system are eventually performed. As we mention below, we draw a line on the conditions under which these two properties can be satisfied and with what cost.

Our approach: We deploy an *incremental* approach in studying the problem. We first assume that there is a centralized authority, called *central scheduler*, that at the beginning of each round informs the processes (that are currently operational) about the tasks that are still pending to be performed, including any new tasks injected in this round. The reason to begin with this assumption is two-fold: (a) The fact that processes have consistent information on the number of pending tasks enables us to focus on identifying the inherent limitations of the problem under processes failures/restarts and dynamic injection of tasks without having to implement information sharing amongst processes. The algorithmic solutions developed under this *information* model are used as building blocks in versions of the problem that deploy weaker information models. Furthermore, lower bound results developed in this information model are also valid for weaker information models. (b) Studying the problem under this assumption has its own independent interest, as the central scheduler can be viewed as an abstraction of a monitor used for monitoring the computation progress and providing feedback to the computing elements. For example it could be viewed as a *master server* in Master-Worker Internet-based computations such as SETI [23] or Pregel [25], or as a *resource broker/scheduler* in Computational Grids such as EGEE [14].

We then limit the information provided to the processes. We consider a weaker centralized authority, called *central injector*, which informs processes, at the beginning of each round, only about the tasks injected in this round and information about which tasks have been performed only in the previous round. We show how to transform solutions for the task performing problem under the model of central scheduler into solutions for the problem under the model

of central injector with the expense of sending a quadratic number of messages in every round. It also occurs that a quadratic number of messages must be sent in some rounds by any correct distributed solution for the considered problem in the model of central injector.

With the gained knowledge and understanding, we then show how processes can obtain common knowledge on the set of pending tasks without the use of a centralized authority. We now assume the existence of a *local injector* that injects tasks to processes without giving them any global information (for example, each process may be injected tasks that no other process in the system has been injected, or only a subset of processes may be injected the same task). The injector can be viewed, for example, as a local daemon of a distributed application that provides local information to the process that is running on. We show that solutions to this more general setting come with minimal cost to the competitiveness, provided that *reliable multicast* [8] is available.

Our results: We now summarize our results. (All results concern deterministic solutions.)

(a) Solutions guaranteeing correctness: For the model of central scheduler, we show a lower bound of OPT$+n/3$ on the pending-tasks competitiveness of any deterministic algorithm, even for algorithms that make use of messages and are designed for restricted forms of crash-restarts patterns. We claim that this lower bound result is valid in all other settings we consider. We then develop the near-optimal deterministic algorithm AlgCS that does not make any use of message exchange amongst processes and achieves OPT $+ 2n$ pending-tasks competitiveness. Using a *generic transformation* we obtain algorithm AlgCI for the model with central injector with the same competitiveness as algorithm AlgCS. Algorithm AlgCI has processes sending messages to each other in every round. Finally, we develop algorithm AlgLI for the model with local injector and we show that it achieves OPT $+ 3n$ pending-tasks competitiveness, under the assumption of reliable multicast. These results are presented in Sect. 3.

(b) Solutions guaranteeing fairness: The issue of fairness is far more complex than correctness; we show that it is necessary and sufficient to assume that when a process restarts it does not fail again in the next at least two consecutive rounds; under this restriction, called 2-*survivability*, we develop fair algorithms AlgCSF, AlgCIF, and AlgLIF in the three considered information models and show that they "suffer" an additional additive surplus of n to their competitiveness, comparing to the algorithms that guarantee only correctness. An interesting observation is that fairness can only be guaranteed in infinite executions, otherwise competitive solutions are not possible. These results are detailed in Sect. 4.

(c) Bounding communication: We show that in the model of central injector and local injector, if processes do not send messages to all other processes, then correctness (and thus also fairness) cannot be guaranteed, unless stronger restrictions are imposed on the crash-restart patterns. This result is detailed in Sect. 5.1.

(d) Non-unit-length tasks: For the above results we assumed that tasks are of unit-length, that is, they require one round to be performed by some process. The situation is even more complex when tasks may not be of unit-length. For the model of central scheduler, we show that if tasks have uniform length $d \geq 1$, that is, each task requires d consecutive rounds to be performed by a process, then a variation of algorithm AlgCS achieves OPT $+ 3n$ pending-tasks competitiveness, under the correctness requirement. We conjecture that similar techniques can be applied to obtain competitive algorithms in the other information models and under the fairness requirement. Then we show that bounded competitiveness is not possible if tasks have different lengths, even under slightly restricted adversarial patterns. These results are given in Sect. 5.2.

The negative results of (c) and (d) give rise to interesting research questions and yield interesting future research directions. These are discussed in Sect. 6.

Related Work. The *Do-All* problem has been studied in several models of computation, including message-passing (e.g., [13, 20]), shared-memory (e.g., [1, 22, 24]), partitionable networks (e.g., [19]), in the absence of communication (e.g., [27]) and under various assumptions on synchrony/asynchrony and failures. As already mentioned, the underlying assumption is that the number of tasks m is *fixed, bounded and known* a priori (as well as the task specifications) by all processes. The *Do-All* problem is considered solved when all tasks are performed, provided that at least one process remains operational in the entire computation (this can be viewed as a simplified version of our *fairness* property). The efficiency of *Do-All* algorithms is measured either as the total number of tasks performed – *work complexity* [13] or as the total number of *available processes steps* [22]. Georgiou et al. [18] considered an iterated version of the problem, where *waves* of m tasks must be performed, one after the other. All task waves are assumed to be known a priori by the processes. Clearly the problem we consider in this work is more general (and harder), as tasks do not come in waves, are not known a priori, and their number might not be bounded. Furthermore, we consider processes crashes

and restarts, as opposed to the work in [18] that considers only processes crashes. Chlebus et al. in [8] considered the *Do-All* problem in the synchronous message-passing model with processes crashes and restarts. In order to obtain a solution for the problem in this setting, they made two modeling assumptions: (a) Reliable multicast: if a process fails while mutlicasting a message, then either all (non-faulty) targeted processes receive the message, or none does, and (b) There is at least one process alive for $k > 1$ consecutive rounds of the computation. In the present paper, as already mentioned, we also require reliable multicast in the model with local injector, and as we discuss in later sections, to guarantee fairness we require a similar restriction on the process living period. Finally, in [19], an online version of the *Do-All* problem is considered where the network topology changes dynamically and processes form disjoint communication groups. In this setting the efficiency (work complexity) of a randomized *Do-All* algorithm is compared with the efficiency of an offline algorithm that is aware a priori of the changes in the network topology. Again, the number of tasks is fixed, bounded and known a priori to all processes.

The notion of competitiveness was introduced by Sleator and Tarjan [29] and it was extended for distributed algorithms in a sequence of papers by Bartal et al. [7], Awerbuch et al. [6], and Ajtai et al. [2]. Several distributed computing problems have been modeled as online problems and their competitiveness was studied. Examples include distributed data management (e.g., [7]), distributed job scheduling (e.g., [6]), distributed collect (e.g., [9]), and set-packing (e.g., [15]).

In a sequence of papers [10, 11, 26] a scheduling theory is being developed for scheduling computations having intertask dependencies for Internet-based computing. The objective of the schedules is to render tasks eligible for execution at the maximum possible rate and avoid gridlock (although there are available computing elements, there are no eligible tasks to be performed). The task dependencies are represented as directed acyclic tasks and the theory has been extending the families of DAGs that optimal schedules can be developed. This line of work mainly focuses on exploiting the properties of DAGs in order to develop schedules. Our work, although it considers independent tasks, focuses instead, on the development of distributed fault-tolerant task performing algorithms and exploring the limitations of online distributed collaboration.

2 Model

Distributed setting. We consider a distributed system consisting of n synchronous, fault-prone, message-passing processes, with unique ids from the set $[n] = \{1, 2, \ldots, n\}$. We assume that processes have access to a global clock. We further assume a fully connected underlying communication medium (that is, each process can directly communicate with every other process) where messages are not lost or corrupted in transit.

Rounds. For a simplicity of algorithm design and analysis, we assume that a single round is split into four consecutive steps: (a) Receiving step, in which a process receives messages sent to it in the previous round; (b) Task injection step, in which new tasks are injected to processes, if any; (c) Local computation step, in which a process performs local computation, including execution of at most one task; and (d) Sending step, in which a process sends messages to other processes as scheduled in the local computation part.

Tasks. Each task specification τ is a tuple $(id,\ \rho,\ code)$, where $\tau.id$ is a positive integer that uniquely identifies the task in the system, $\tau.\rho$ corresponds to the round number that the task was first injected to the system to some process (or set of processes), and $\tau.code$ corresponds to the computation that needs to occur so that the task is considered completed (that is, the computational part of the task specification that is actually performed). *Unless otherwise stated, c.f., Sections 5.2 and 6*, we assume that it takes one round for each task to be performed, and it can be performed by any process which is alive and knows the task specification.

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

Adversary. We assume an adaptive and omniscient adversary that can cause crashes, restarts and task injections. We define an adversarial pattern $\mathcal{A}$ as a collection of crash, restart and injection events caused by the adversary. A $crash(r, i)$ event specifies that process i is crashed in round r. A $restart(r, i)$ event specifies that process i is restarted in round r; it is understood that no $restart(r, i)$ event can take place if there is no preceding $crash(r', i)$ event such that $r' < r$.[1] Finally an $inject(r, i, \tau)$ event specifies that process i is injected the task specification τ in round r.

We say that a process i is *alive* in a round r if the process is operational at the beginning of the round and does not fail by the end of the round (a process that restarts at a beginning of a round and does not fail by the end of the round is also considered alive in that round). We assume that when the adversary injects tasks in a given round, it injects a finite number of tasks.

Restarts of processes: We assume that a restarted process has knowledge of only the algorithm being executed and the ids of the other system processes (but no information on which processes are currently alive). Algorithmically speaking, once a process restarts, it waits to receive messages or to be injected tasks. Then it knows that a new round has begun and hence it can smoothly start actively participating in the algorithm. For the ease of analysis and better clarity of result exposition we simply assume that processes are restarted at the beginning of a round – but processes could fail at any point during a round. We also assume that a process that restarts in the beginning of round r receives the messages sent to it (if any) at the end of round $r - 1$.

Admissibility: We say that an adversarial pattern is *admissible*, if

(a) in every round there is at least one alive process; in case of finite executions, all processes alive in the last round are crashed right after this round (in other words, a finite execution of an algorithm ends when all processes are crashed); and

(b) a task τ that is injected in a given round is injected to at least one alive process in that round; that is, the adversary gives some window of opportunity for task τ to either be performed in that round or other processes to be informed about this task.

Condition (a) is required to guarantee some progress in the computation. To motivate condition (b), consider the situation where a process in a given round is injected a task τ (and this is the only process being injected task τ) and then the process immediately crashes. No matter of the scheduling policy or communication strategy used, task τ cannot be performed by any algorithm; with condition (b) we exclude the consideration of such uninteresting cases; these tasks are not taken into consideration, neither for correctness, nor for performance issues. From this point onwards we only consider admissible adversarial patterns.

Restricted classes of adversaries. As we show later, some desired properties of task performing algorithms, such as fairness, may not be possible to achieve in general executions under any admissible adversarial pattern. In such cases, we also consider a natural property that restricts the power of adversary, called *t-survivability*: Every awaken[2] process must stay alive for at least t consecutive rounds, where $t \geq 1$ is an integer.

Information models. In regards to the distribution of injected tasks, as discussed in Section 1, we study three settings:

(i) *central scheduler*: in the beginning of each round it provides all operational processes with the current set of unperformed tasks' specifications;

(ii) *central injector*: in the beginning of each round it provides all operational processes with specifications of all newly injected tasks, and also confirmation of tasks being performed in the previous round, i.e., for round r, it informs all operational processes of the tasks injected in this round and the tasks that have been successfully performed in round $r - 1$;

(iii) *local injector*: in the beginning of each round it provides each operational process with specifications of tasks injected into this process; a task specification may be injected to many processes in the same round.

[1] With the exception of a process first entering the computation; not all processes may be awake at the beginning of the computation.

[2] This includes restarted processes and processes first entering the computation.

Correctness and fairness. We consider two important properties of an algorithm: correctness and fairness.

Correctness of an algorithm: An algorithm is *correct* if for any execution of the algorithm under an *admissible adversarial pattern*, for any injected task and any round following the injection time, there is a process alive in this round that stores the task specification, unless the task has been already performed. Observe that this property does not guarantee eventual performance of a task.

Fairness of an algorithm: We call an *infinite* execution of an algorithm under an adversarial pattern *fair execution* if each task injected during the execution is eventually performed. We say that an algorithm is a *fair algorithm* if every infinite execution of this algorithm is fair.[3] In other words, this property requires correctness, plus the guarantee that each task is eventually performed in any infinite execution of an algorithm. Observe that the greedy offline algorithm described above is fair.

Efficiency measures. *Per round pending-tasks complexity*: Let P_r denote the total number of pending tasks at the beginning of round r, where by *pending task* we understand a task which has been already injected to some process (or a set of processes) but not yet performed by any process[4]. Then the per round pending-tasks complexity is defined as the maximum P_r over all rounds (supremum in case of infinite computations).

In case of competitive analysis, we say that the competitive pending-tasks complexity is $f(\mathrm{OPT}, n)$, for some function f, if and only if for every adversarial pattern $\mathcal{A}$ and round r the number of pending task in round r of the execution of the algorithm against adversarial pattern $\mathcal{A}$ is at most $f(\mathrm{OPT}(\mathcal{A}, r), n)$, where $\mathrm{OPT}(\mathcal{A}, r)$ is the minimum number of pending tasks achieved by an off-line algorithm, knowing $\mathcal{A}$, in round r of its execution under the adversarial pattern $\mathcal{A}$. In the classical competitiveness methodology function f needs to be linear with respect to the first coordinate, however as we will show, sometimes more accurate functions can be produced for the problem of distributed task performance.

Observe that the above definition allows for the optimum complexity of two different rounds to be met by two different optimum algorithms. However a simple greedy algorithm scheduling different pending tasks (with largest possible pending time) to different alive processes at each round is optimal from the perspective of any admissible adversarial pattern $\mathcal{A}$ and any round r (recall that to specify the optimum algorithm we can use the knowledge of $\mathcal{A}$).

For the sake of more sensitive bounds on competitiveness of algorithms, we consider subclasses of adversarial paterns achieving the same worst-case performance in terms of the optimum solution. These classes are especially useful for establishing sensitive lower bounds. We say that an adversarial pattern $\mathcal{A}$ is (k, r)-*dense* if $\mathrm{OPT}(\mathcal{A}, r) = k$. A pattern $\mathcal{A}$ which is (k, r)-dense for some round r is called k-*dense*.

In Section 5.1 we also study the message complexity of solutions to the task performing problem. Specifically we consider *per-round message complexity*, defined as the maximum number of point-to-point messages sent in a single round of an execution of a given algorithm, over all executions and rounds.

3 Solutions Guaranteeing Correctness

In this section we study the problem focusing on developing solutions that guarantee correctness, but not necessarily fairness (this property is studied in Section 4). We consider unit-length tasks (non-unit-length tasks are discussed in Section 5.2). We impose no restriction on the number of messages that can be sent in a given round; for example processes could send a message to every other processes, in every round (the issue of restricted communication is studied in Section 5.1). The results of this section are obtained in the most general of considered settings: the upper bounds hold against any admissible adversary, while the lower bounds hold even in the presence of a restricted adversary satisfying t-survivability, for any t.

[3] It is not difficult to see that the adversary can form *finite* executions in which not all tasks can be performed, not even by the offline algorithm.

[4] If a task was performed by some process, but the adversary did not provide the possibility to this process to inform another process or a central authority (scheduler or injector) — e.g., the process is crashed as soon as it performs the task — then this task is not considered performed.

3.1 Central Scheduler

We first show that all algorithms require a linear additive factor in their competitive pending-tasks complexity. This bound holds for all settings considered in this work, as the central scheduler is the most restrictive one.

Theorem 3.1. *Every algorithm has competitive pending-tasks complexity of at least $k + n/3$ against some k-dense adversarial pattern satisfying t-survivability, for every non-negative integers k, t.*

Proof: Consider an algorithm Alg. Fix a round r such that all processes are alive at the beginning of the round and have been alive for more than t rounds (hence t-survivability is satisfied) and there are no pending tasks, neither for Alg nor for the optimum offline solution (for example, the adversary, up to round $r > t$ failed no processes and was injecting in every round as many tasks as Alg would be able to perform all of them by the end of the round). Now consider the following adversarial pattern $\mathcal{A}$ for round r: $2n/3$ tasks are injected and $n/3$ processes are crashed at the beginning of the round.

Since the optimum offline solution knows a priori the processes that will be crashed, it allocates the tasks to the processes that will not fail. Hence $\text{OPT}(\mathcal{A}, r + 1) = 0$. Now, it is not difficult to see that the best allocation that Alg can obtain is to have a different task to be allocated to $2n/3$ processes and each of the remaining $n/3$ processes being allocated a task that has already been allocated to another process. It follows that there exist at least $n/3$ tasks that are uniquely allocated to a process (that is, at most one process has been allocated such task). The adversary crashes these processes and hence $\text{Alg}(\mathcal{A}, r + 1) \geq n/3$.

In round $r + 1$ the adversary injects $2n/3 + k$ new tasks, for some $k \in \mathbb{Z}^+$. The adversary fails no process. It follows that both the optimum offline solution and Alg may perform $2n/3$ different tasks (each alive process performs a different task) and hence $\text{OPT}(\mathcal{A}, r + 2) = k$ and $\text{Alg}(\mathcal{A}, r + 2) \geq k + n/3$. Observe that $\mathcal{A}$ is k-dense for round $r + 2$.

Finally observe that even if processes in algorithm Alg exchange some information (e.g., regarding their state or knowledge) amongst them in every round, the described adversarial pattern results in the same competitiveness, as the arguments above work under the assumption that processes know the whole execution in the previous rounds. □

Next, we show that the following simple algorithm, specified for a process i and a round r, is near-optimal. Observe that the algorithm does not require sending messages between processes.

Algorithm AlgCS(i, r)

- Get set of pending task specifications from the scheduler.
- Rank the task specifications in incremental order, based on the task id ($\tau.id$ for a task specification τ).
- Perform task with rank $i \bmod n$.

Theorem 3.2. *Algorithm AlgCS achieves competitive pending-tasks complexity of at most OPT $+ 2n$ against any admissible adversary.*

Proof: Suppose, to the contrary, that algorithm AlgCS has more than $\text{OPT}(\mathcal{A}, r) + 2n$ pending tasks at the end of some round r of some execution of the algorithm under some adversarial pattern $\mathcal{A}$ in which the optimum number of pending tasks is $\text{OPT}(\mathcal{A}, r)$. W.l.o.g. assume that r is the first such round in the execution under adversarial pattern $\mathcal{A}$. Let r^* be the largest round number before r at the end of which there are smaller than $\text{OPT}(\mathcal{A}, r^*) + n$ pending tasks in the algorithm (it is possible that r^* is the first round of the execution). Hence, in the time interval $[r^* + 2, r]$ the scheduler informs processes about a set of at least n tasks, the same set for all processes (note that $r^* + 2 \leq r$, since it would take at least two rounds for the competitiveness to grow by n). Therefore, algorithm AlgCS assures that all alive processes in a round in this period perform pairwise different tasks, which means that the number of newly performed tasks is the same as by any optimum algorithm against $\mathcal{A}$ in this round. Observe also that the difference of performed tasks in round $r^* + 1$ by the optimum algorithm and algorithm AlgCS is at most $n - 1$; since there is at least one alive process (admissibility property) then the optimum performs at most n different tasks while AlgCS performs at least one task. These two facts imply that the number of tasks remaining at the end of round r of the execution of the algorithm is *smaller than*

$$(\text{OPT}(\mathcal{A}, r^*) + n) + (\text{OPT}(\mathcal{A}, r) - \text{OPT}(\mathcal{A}, r^* + 1))$$

$$\leq (\text{OPT}(\mathcal{A}, r^*) + n) + (\text{OPT}(\mathcal{A}, r) - ((\text{OPT}(\mathcal{A}, r^*) - (n-1)))$$
$$= \text{OPT}(\mathcal{A}, r) + 2n - 1 \ .$$

This is a contradiction, which concludes the proof of the theorem. □

Remark 3.3. *Since the lower bound of Theorem 3.1 holds even for algorithms that have the processes exchange messages in every round, it follows that one cannot hope to achieve much better competitiveness if algorithm AlgCS (or some other algorithm) uses the full communication paradigm. As we will also stress later, this lower bound automatically holds for the models with central/local injector, as the feedback given to the algorithm in these two models is a subset of the feedback that needs to be provided by central scheduler.*

3.2 Central Injector

We now relax the information given to the processes in the beginning of every round by considering the weaker information model of central injector. We first show how to *transform* an algorithm specified for the setting with central scheduler, call it *source algorithm*, into an algorithm specified for the setting with central injector, call it *target algorithm*. The transformation maintains all local variables used by the source algorithm and sends the same messages, but now additional local variables are used and messages may contain additional information, required by the processes in the target algorithm in order to obtain the same set of pending tasks (under the same adversarial pattern) as the one that the central scheduler provides by default to the processes in the source algorithm.

The main structure of a generic algorithm, call it GenCS, specified for the setting with central scheduler is as follows (for a process i and round r):

Source Algorithm GenCS(i, r)

- Get set $\mathcal{P}$ of pending task specifications from the scheduler.
  Receive messages by each process j that sent a message in the previous round containing information x_j.
- Based on $\mathcal{P}$ and each received information x_j deploy the scheduling policy $\mathcal{S}$ to perform a task.
- Send a message with information x_i to all other processes.

We now present the target algorithm, call it GenCI, which is obtained when we deploy our transformation, call it $tranCStoCI$, to the source algorithm GenCS. The text in bold annotates the elements added from $tranCStoCI$ (these elements essentially specify the transformation).

Target Algorithm GenCI(i, r)

- **Get set $\mathcal{N}$ of specifications of newly injected tasks and set $\mathcal{D}$ of tasks confirmed as done in round $r-1$, from the central injector.**
  Receive messages by each process j that sent a message in the previous round containing information x_j**, and $\mathcal{P}_j$. Let $\mathbf{R} = \{j :$ received a message from j in this round$\}$.**
- **Update local set $\mathcal{P}_i$ of pending tasks as follows: $\mathcal{P}_i = \bigcup_{j \in R \cup \{i\}} \mathcal{P}_j \cup \mathcal{N} \setminus \mathcal{D}$.**
- Based on $\mathcal{P}_i$ and each received information x_j deploy the scheduling policy $\mathcal{S}$ to perform a task.
- Send a message with information x_i **and** $\mathcal{P}_i$ to all other processes.

It is evident that algorithm GenCI continues to maintain the variables of GenCS and sends the same messages as algorithm GenCS (but with more information). What remains to show is that the set of pending tasks used in the scheduling policy $\mathcal{S}$ in a given round is the same for both algorithms.

Lemma 3.4. *For any given round r, the set of pending tasks used in the scheduling policy $\mathcal{S}$ is the same in the executions of algorithms GenCS and GenCI formed by the same adversarial pattern.*

Proof: Consider two parallel executions of the algorithms under the same adversarial pattern. The proof proceeds by induction on rounds. Consider the base case (round 1). In algorithm GenCS the central scheduler provides to all alive processes the set of pending tasks, which is essentially the number of newly injected tasks. This information is also provided by the central injector in algorithm GenCI ($\mathcal{P} = \mathcal{N}$ and $\mathcal{D} = \emptyset$). Since no messages are received, the claim of the lemma holds.

Assume that the claim of the lemma holds for $r-1$, we show that it also holds for round r. (Note that this claim also implies that the local sets of pending tasks of the processes in algorithm GenCI are the same, since they are the same with the ones in GenCS, which by definition are the same.) By inductive hypothesis, the task chosen to be performed by each process that is alive in the task performance step of round $r-1$ is the same in both algorithms, as the scheduling policy is applied on the same information. Since the same adversarial pattern is applied, a process that does not perform its chosen task in round $r-1$ of algorithm GenCS will also not perform it in round $r-1$ of algorithm GenCI. Therefore, the set of tasks performed in round $r-1$ is the same for both algorithms. Furthermore, the processes that manage to send a message at the end of round $r-1$ in the one algorithm are the same as in the other algorithm. The processes in algorithm GenCI send, additionally to the information x, their set of pending tasks $\mathcal{P}$. By inductive hypothesis, this set is the same to all processes at the sending phase of round $r-1$ (since it was the same in the task performing phase and it does not change after that). Due to the admissibility assumption, there must be at least one process that manages to send a message to all other processes in round $r-1$.

In the beginning of round r, in algorithm GenCS, the central scheduler provides to all alive processes the set of pending tasks. This set includes the older tasks that remain pending by the end of round $r-1$ and the newly injected tasks. The pending tasks are the tasks that were pending at the scheduling step of round $r-1$ minus the tasks that were performed during the task performing step of that round. The set of newly injected tasks ($\mathcal{N}$) and the set of tasks that were performed in round $r-1$ ($\mathcal{D}$) are provided by the central injector to the processes that are alive at the beginning of round r in algorithm GenCI. The set of pending tasks of round $r-1$ is included in the message sent in round $r-1$ by a alive process (per admissibility there is at least one). By induction this set is the same as the one in algorithm GenCS and hence it follows that in the update step of algorithm GenCI in round r, the processes will obtain the same set of pending tasks as the processes in algorithm GenCS. □

Consider algorithm AlgCS of Section 3.1. This algorithm is a specialization of algorithm GenCS where the x_i's are $null$ and the scheduling policy $\mathcal{S}$ is simply ranking the tasks in $\mathcal{P}$ in incremental order (based on their ids) and having process i perform task with rank i mod n. Let **Algorithm AlgCI** be the algorithm resulting by applying the transformation $TranCStoCI$ to algorithm AlgCS. Then, from Lemma 3.4 and Theorem 3.2 we get:

Theorem 3.5. *Algorithm AlgCI achieves competitive pending-tasks complexity of at most OPT $+ 2n$ against any admissible adversary.*

Remark 3.6. *The lower bound stated in Theorem 3.1 (Section 3.1) obtained for central scheduler trivially holds also for central injector, as the feedback given to the algorithm in the latter setting is a subset of the feedback from the former setting, and, for any adversarial pattern, the optimum solution is the same in both settings (it is for example the greedy offline algorithm described in Section 2). Therefore, we may conclude that algorithm AlgCI is near-optimal.*

3.3 Local Injector

In this section we consider the local injector model. Consider algorithm AlgLI, specified below for a process i and a round r. In each round r, each process i maintains two sets, *new* and *old*. Set *new* contains all new tasks injected to this process in this round. Set *old* contains older tasks that the process knows they have been injected in the system (not necessarily to this process) but have not been confirmed as done.

The following lemma states that the information on injected tasks is not lost, but it is propagated in the system with a round of delay.

Lemma 3.7. *In any execution of algorithm AlgLI, the tasks injected to the processes in round r are learned by all processes that are alive at the beginning of round $r+1$, under any adversary.*

Algorithm AlgLI(i, r)

- Get specifications of newly injected tasks from local scheduler and store them in set new (remove any older information from this set).
  Receive messages sent (if any) by other processes in round $r - 1$.
- Update set old based on received messages: the new set *old* is the union of all the received sets *old* and *new* minus the tasks that have been reported in the current round as done in the previous round.
- Perform a task based on the following scheduling policy: if set $old \neq \emptyset$ then rank tasks in old incrementally based on their ids and perform task with rank $i \mod |old|$. Otherwise, and if new is not empty, then rank the tasks in new incrementally based on their ids and perform task with the smallest rank.
- Send to all other processes sets new, old and the task id of the performed task.

Proof: Fix a round r. Let $\mathcal{L}_r$ denote the set of processes that are alive for the whole of round r; from the definition of admissibility (clause (b)), from the tasks injected in round r, we only focus on the tasks injected to the processes in this set. We denote by $I_{r,i}$ the set of tasks injected to process i in round r, $i \in \mathcal{L}_r$. Then $I_r = \bigcup_{i \in \mathcal{L}_r} I_{r,i}$ is the set of tasks injected in the system in round r.

Per algorithm AlgLI each process $i \in \mathcal{L}_r$ sends to all other processes set $I_{r,i}$ (along with some other information). Since these processes are alive in the whole round and there is at least one live process in round $r + 1$ (admissibility restriction), at least one process learns the whole set I_r at the beginning of round $r + 1$. Hence, the information on injected tasks is not lost and it is propagated in the system with a round of delay. This completes the proof. □

The following lemma shows that at the beginning of each round processes have consistent information on the set of pending tasks. Here *reliable multicast* is assumed [8]: if a process crashes while multicasting a message, then either all targeted processes (that are alive) receive the message or none does.

Lemma 3.8. *In any execution of algorithm AlgLI assuming reliable multicast, the processes that are alive at the beginning of each round r (before the injection step) have the same information on the set of pending tasks.*

Proof: We proceed by induction on rounds. The base case holds trivially. Assuming that the claim holds for round $r - 1$, we show that it also holds for round r.

By induction hypothesis, all processes that are alive at the beginning of round $r - 1$ have the same information on pending tasks. In particular they have the same set *old*. From these processes, the ones that are alive for the whole round may also be injected new tasks. We denote the set containing these processes by $\mathcal{L}_{r-1}$. Also let $I_{r-1,i}$ be the set of tasks injected to process $i \in \mathcal{L}_{r-1}$ in round $r - 1$ ($I_{r-1,i}$ can be empty). Per algorithm AlgLI, the processes in round $r - 1$, perform a task (using the specified scheduling policy) and send their sets *new*, *old*, and the task id of the task they performed in this round (note that for each process $i \in \mathcal{L}_{r-1}$, *new* = $I_{r-1,i}$).

First we consider the processes in $\mathcal{L}_{r-1}$. Since these processes are alive in the whole round, their messages are received by all processes that are alive at the beginning of round r: let set *LiveBegNextRound* denote these processes. Per Lemma 3.7 all processes in *LiveBegNextRound* learn all the new tasks injected in the previous round (and hence, have consistent information with respect to these tasks). Furthermore, the processes in *LiveBegNextRound* receive the sets *old* from the processes in $\mathcal{L}_{r-1}$ (they have the same set *old*) and the tasks performed by them in round $r - 1$. Now, consider the processes that were alive at the beginning of round $r - 1$ but failed during the round (by admissibility, we do not care about the tasks injected to these processes in round $r - 1$, unless these tasks were also injected to processes in $\mathcal{L}_{r-1}$). These processes could have performed a task before failing. If such a process fails before sending a message, then no harm is done. But even if such a process fails while sending a message, then the reliable multicast assumption guarantees that the processes in *LiveBegNextRound* either all receive this message (and hence the information that a task has been performed) or none does. It straightforwardly follows that when the processes in *LiveBegNextRound* take the union of the received sets *old* and *new* and remove the tasks reported to be performed in the previous round they all form the same updated set *old*. This completes the proof. □

To show the correctness of algorithm AlgLI it remains to show that no task is "lost".

Lemma 3.9. *In any execution of algorithm AlgLI, assuming reliable mutlicast, if a task specification is no longer in the system, then it is the case that the task has been performed by some process.*

Proof: Per Lemma 3.7 we have that no new task is lost until at least the next round that it was injected. So it remains to show that during the update phase of a round r, if a process removes a task τ for its local set *old*, this is because it has been reported by a process that this task has been performed in round $r-1$. We proceed by induction on rounds.

The base case (round 1) holds trivially, as all processes have empty sets *old*. Assume that the claim holds up to round $r-1$ and prove for round r. Fix a process i that is alive at the beginning of round r and will remain alive through the round (there is at least one such process due to the admissibility restriction). Per Lemma 3.8 it is immaterial whether i was alive or not in round $r-1$ (all processes have the same information on pending tasks). Consider a task τ. We consider two cases:
(a) Task τ was injected to some process(es) in round $r-1$. Hence, τ was included in the set *new* of the process(es) it was injected at, and since at least one of these processes were alive, τ is included in process' i set *old* at the beginning of round r. However, if i also receives a report that τ was performed then it removes it from τ. Observe that only the process(es) that τ was injected at, could perform it in round $r-1$ (as only these processes are aware of τ) and given that processes do not lie, τ was indeed performed. If i does not receive such report, then τ is included in process' i set *old* in round r, and the existence of τ is propagated (by at least process i) to the next round.
(b) Task τ is a task injected in a round prior to $r-1$. From Lemmas 3.7 and 3.8 it follows that τ was included in the set *old* of at least one process and by inductive hypothesis τ was not performed until the beginning of round $r-1$. Hence τ will be included in all sets *old* received by process i at the beginning of round r. And following the same reasoning as in case (a), it follows that i will remove τ only if it has learned that τ was performed by some process in round $r-1$. This completes the proof. □

We now show the competitiveness OPT $+ 3n$ of algorithm AlgLI, as stated in the theorem that follows:

Theorem 3.10. *Algorithm AlgLI, assuming reliable multicast, is correct and near-optimal; more precisely, it achieves competitive pending-tasks complexity of at most OPT* $+ 3n$ *against any adversary.*

Proof: Correctness follows directly from Lemma 3.9. The proof of competitiveness is similar to the proof of Theorem 3.2. The key difference lies in the fact that under the central scheduler processes are informed about the newly injected tasks in the same round as opposed to the local injector that it takes an additional round (per Lemma 3.7). Note that the optimum offline solution does not suffer from this delay, as it knows the injection pattern a priori. As it turns out, this round delay does not affect the competitiveness of the algorithm by more than an additive factor of n. We begin with the following claim.
Claim 1: *If there are at least n pending **old** tasks in the beginning of a round r, then in round r all alive processes perform pairwise different tasks.*
We now prove Claim 1. From Lemma 3.8 we have that all processes that are alive in round r have the same set old and per the thesis of the claim, $|old| \geq n$. Regardless of whether new tasks are injected in round r, due to the scheduling policy and common knowledge of old, all alive processes perform different tasks. This completes the proof of the claim.

We now make another claim.
Claim 2: *In any round r, the number of pending **old** tasks under any adversarial pattern $\mathcal{A}$ is at most OPT$(\mathcal{A}, r-1)+2n$, while OPT$(\mathcal{A}, r)$ contains at least all new tasks pending except at most n and the algorithm has all new tasks pending.*

From Claim 2 it follows that the algorithm has at most OPT$(\mathcal{A}, r) + 3n$ total pending tasks (both old and new together) and this competitiveness cannot grow any further: in round $r+1$ there are more than n pending tasks, so by Claim 1 the algorithm performs as many task as the optimum offline solution (so the competitiveness does not increase regardless of the number of injected tasks); if the number of old pending tasks drops below OPT$(\mathcal{A}, r+1) + 2n$, then we go back to the statement of Claim 2 for round $r+1$. Therefore it remains to prove Claim 2.

Assume, to arrive at a contradiction, that there is a round r^* of an execution of the algorithm under some adversarial pattern $\mathcal{A}$ in which the number of pending **old** tasks by the end of the round is bigger than OPT$(\mathcal{A}, r^*) + 2n$. Moreover, let r^* denote the first such round. Let r be the oldest round before r^* such that the number of pending **old** tasks by the end of the round is at most OPT$(\mathcal{A}, r)+n$ (it is possible that r is the first round of the execution). It follows that in round $r+1$ the number of pending **old** tasks is bigger than OPT$(\mathcal{A}, r+1)+n \geq n$ and smaller than OPT$(\mathcal{A}, r+1)+2n-1$,

by the fact that the algorithm performs at least one task (admissibility restriction). It follows that $r+1 < r^*$. Then, in the time interval $[r+2, r^*]$, containing at least one round, the number of pending **old** tasks at the end of each round is bigger than n, and therefore by Claim 1 it follows that the number of tasks performed by the algorithm is the same as the number of tasks performed by the optimum offline solution. Thus, at the end of round r^* the number of pending **old** tasks is upper bounded by

$$(\text{OPT}(\mathcal{A}, r+1) + 2n - 1) + (\text{OPT}(\mathcal{A}, r^*) - \text{OPT}(\mathcal{A}, r+1)) = \text{OPT}(\mathcal{A}, r^*) + 2n - 1 \ ,$$

which contradicts the assumption that this number is bigger than $\text{OPT}(\mathcal{A}, r^*) + 2n$. This completes the proof of Claim 2 and the proof of the theorem. □

Remark 3.11. *The lower bound stated in Theorem 3.1 (Section 3.1) obtained for central scheduler trivially holds also for local injector, as the feedback given to the algorithm in the latter setting is a subset of the feedback from the former setting, and, for any adversarial pattern, the optimum solution is the same in both settings (it is for example the greedy offline algorithm described in Section 2). Therefore, we may conclude that algorithm AlgLI is near-optimal.*

4 Solutions Guaranteeing Fairness

We now turn our attention to the much challenging problem of guaranteeing fairness. Recall from Section 2 that for fairness we consider only infinite executions and for such executions there is always a fair (offline) algorithm.

4.1 Central Scheduler

We first demonstrate that the issue of fairness is much more involved than correctness. Consider the following simple fair algorithm LIS: each process performs the Longest-In-System task, and in case of a tie it chooses the one with the smallest task id.

Fact 4.1. *Algorithm LIS has unbounded pending-tasks competitiveness under any adversary, even for the restricted one satisfying t-survivability, for any $t \geq 1$.*

Proof: Consider the following adversarial pattern: all processes are initially alive and the adversary injects n tasks in every round and crashes no processes. The optimum offline solution is aware of this pattern and hence it performs all tasks in every round. However, algorithm LIS performs exactly one task in every round, and hence the number of pending tasks increases by $n-1$ in each round, yielding unbounded competitiveness. Note that since the described adversarial pattern involves no process crashes (or restarts) the claimed competitiveness holds against any admissible adversary, even the one satisfying t-survivability, for any t. □

The above shows that a fair algorithm not only needs to have some provision in eventually performing a specific task but it also needs to guarantee progress when a large number of tasks is pending. Furthermore, we show that admissibility alone is not enough to guarantee both fairness and bounded competitiveness.

Theorem 4.2. *For any fair algorithm and any integer $y > 0$, there is a round r and an admissible, adversarial pattern $\mathcal{A}$ such that the algorithm has more than $y \cdot (\text{OPT}(\mathcal{A}, r) + 1)$ pending tasks at the end of round r.*

Proof: Fix a fair algorithm Alg. The strategy of the adversary repeats cyclically the following parts: Let x be the number of tasks that are pending in the execution of the optimum offline algorithm on the already defined parts of the adversarial pattern. In the beginning of each round, the adversary chooses some $\max\{n-x, 0\}$ non-injected tasks and injects them into the system. In the first round of the constructed part of the pattern, the adversary "simulates" the algorithm to check which of the pending tasks (including the newly injected ones) would be performed by each process if it was alive in this round (under the assumption that it also knows its own history of the previous parts of the execution and receives messages potentially sent in the preceding round, as well as the feedback from the central scheduler). There are two cases.

Case 1: If all processes would do the same task, the adversary awakes/restarts all processes that were not alive in the previous round, and finishes the construction of the current part of its pattern.
Case 2: If at least two different tasks would be scheduled, the adversary chooses the task among these tasks for which the smallest number of processes could perform it in the current round; we call it a *critical task*. The critical task is fixed for the whole constructed part of the adversarial pattern. Then the adversary crashes all processes which were alive in the previous round and would like to perform the critical task in the current round, while assuring that all other processes (i.e., processes that do not want to perform the critical task) are alive (if they are alive it keeps them alive, if they are crashed, it restarts them). In the next round, the adversary repeats injecting new tasks according to the rule specified in the beginning of the construction, and also simulates the algorithm for each process to check which task would be scheduled to, assuming the process is alive. Then if all processes declare to perform the critical task, the adversary applies the same rule as in Case 1 (i.e., assures that all processes are alive and finishes the construction of the current part of its pattern). Otherwise, it assures that all processes that do not want to perform the critical task are alive (restarts them if they are not), while crashing all that were alive in the previous round and would like to perform the critical task. This concludes the construction of a single part of the adversarial pattern.

First, we argue that there is an infinite number of consecutive parts. Indeed, observe that each part must have bounded length, since otherwise the critical task of this part would not be performed during the execution of the algorithm, contradicting the fact that the algorithm is fair. Second, we prove that after executing the algorithm by the end of part j of the constructed adversarial pattern, the competitiveness is at least OPT $+ j$. To see this, note that OPT is always at most n, and in each round there is at least as many pending tasks as the number of alive processes in this round in the execution of the optimum algorithm, by the rule of injecting tasks. It follows that in each round of the execution of the optimum algorithm alive processes perform pairwise different tasks, i.e., no process step is wasted for idling or performing the same task twice or more. On the other hand, in each part of the execution of algorithm Alg corresponding to some part of the adversarial pattern, there is a round in which at least two processes perform the same task. These observations imply that the additive overhead above OPT grows by at least one after each part.

Since for each j we have a round in which the number of pending tasks is at least OPT $+ j$, and moreover because OPT $\leq n$, we get that for each integer $y > 0$ there is a round in the execution of algorithm Alg under the constructed adversarial pattern such that the number of pending tasks is at least $y \cdot (\text{OPT} + 1)$. □

Note that Theorem 4.2 implies that the algorithms presented in Section 3 are not fair. Therefore, in order to achieve both fairness and competitiveness, one needs to consider some restrictions to the adversary. It can be easily verified that the impossibility statement in Theorem 4.2 holds even if 1-survivability is assumed. As it turns out, it is enough to assume 2-survivability to be able to obtain fair and competitive algorithms.

Consider algorithm AlgCSF specified below for process i and round r. Each process i maintains a variable age that counts the number of rounds that i has been alive since it last restarted. A restarted process has $age = 0$, and it increments it by one at the end of each local computation part. For simplicity, we say that in round r process is in age x if it was alive for the whole x rounds, i.e., its age is x in the beginning of the round. Processes exchange these variables, so, for reference reasons, we will be denoting by $age_r(j)$ the age that process i knows that j has in round r (in other words, this is the age j reports to i at the end of round r).

We begin to show that algorithm AlgCSF is fair, under the assumption of 2-survivability.

Lemma 4.3. *If in a given round r, τ_{old} is the oldest pending task in the system (has rank 1) and there is at least one process with $age_r = 1$, then τ_{old} is performed by the end of round r.*

Proof: If in round r there are more than $2n$ pending tasks for algorithm AlgCSF, then one of the processes in the set $ASure$ (there is at least one such process by assumption) will perform τ_{old}. Note that processes construct the same set $ASure$ since these processes were alive in the previous round (otherwise their age would not be equal to 1 in the beginning of local computation in round r) and hence their messages is received by all processes alive at the beginning of round r (this includes the processes in $ASure$). If there are at most $2n$ pending tasks, then all alive processes (there is at least one – the one with $age_r = 1$) will perform τ_{old}. □

Observe that if in round r there is no process with age 1 but there is at least one with age 0, then even if τ_{old} is not performed in round r, by Lemma 4.3 it will be performed in round $r + 1$. Hence it remains to show the following.

Algorithm AlgCSF(i,r)

- Get pending tasks from central scheduler and messages sent (if any) in round $r-1$.
- Rank pending tasks *lexicographically*: first based on their pending period (older tasks have smaller rank) and then based on their task ids (incremental order).
- Based on received messages, construct set $ASure$ by including all processes j with $age_r(j) = 1$. If $age = 1$, then i includes itself in the set. /* Processes do not send messages to themselves, but they of course know the value of their local variable age. */
- If the number of pending tasks is larger than $2n$ then
  - If $ASure \neq \emptyset$ then
    - If $age \neq 1$ then perform task with rank $n + i$.
    - Else rank processes in $ASure$ based on their ids and perform task with rank $rank(i)_{ASure}$ (i.e., ith task in set $ASure$).
  - Else [$ASure = \emptyset$]
    - If $age \neq 0$ then construct set $Recved$ by including all processes from whom a message was received at the beginning of the round. Process i includes itself in this set. Then rank processes in set $Recved$ lexicographically, first based on their age and then based on id (increasing order). If $rank(i)_{Recved} = 1$ then perform task with rank 1, otherwise perform task with rank $i + 1$.
    - Else perform task with rank $i + 1$.

  Else perform task with rank 1.
- Set $age = age + 1$.
- Send age to all other processes as the value of variable $age_{r+1}(i)$.

Lemma 4.4. *If in round r all alive processes are of $age > 1$ ($ASure = \emptyset$) and τ_{old} is the oldest task in the system, then τ_{old} will be performed by round $r + 2n$ at the latest.*

Proof: If in round r there are at most $2n$ pending tasks then all alive processes (there must be at least one per admissibility restriction) are allocated to perform τ_{old}, so it is performed in round r.

So, the adversary must maintain the number of pending tasks above $2n$ to prevent the performance of τ_{old}. Recall that tasks are ranked lexicographically, first based on their seniority, so τ_{old} has rank 1. We argue that the adversary cannot delay the performance of τ_{old} by more than $2(n-1)+1$ consecutive rounds in which there are more than $2n$ pending tasks and all alive processes are of $age > 1$. For contradiction, assume that it can. Note that in this period no process is restarted (otherwise in the next round of the process' restart it performs τ_{old} and the adversary, due to 2-survivability, it cannot crash this process), and at most $n-1$ processes may crash. By the pigeonhole principle (applied on number of rounds and number of crashes) there are two consecutive rounds $r', r'+1$ in which no process is crashed. It follows that in round $r'+1$ all alive processes have the same list $Recved$. All processes in this list are alive in round $r'+1$, hence the first in this list performs the oldest task. This is a contradiction. □

Lemmas 4.3 and 4.4 yield fairness of algorithm AlgCSF:

Theorem 4.5. *Algorithm AlgCSF is a fair algorithm under any 2-survivability adversarial pattern.*

It remains to show the competitiveness of algorithm AlgCSF, and this we show against *any* admissible adversarial pattern (unlike fairness, which is guaranteed if the pattern satisfies 2-survivability).

Theorem 4.6. *Algorithm AlgCSF achieves competitive pending-tasks complexity of at most OPT + $3n$ against any admissible adversary.*

Proof: Note that if there are at most $2n$ pending tasks in the beginning of a round r then, by admissibility and algorithm specification, exactly one task is performed by all alive processes. We now investigate the situation when there are more than $2n$ tasks.

Claim: *If there are are more than $2n$ pending tasks in the beginning of a round r then in round r all alive processes perform pairwise different tasks.*

We proceed to prove the claim. We first consider the case where set $ASure \neq \emptyset$. The processes with $age = 1$ form a consistent set $ASure$ (since they were all alive in the previous round) and perform different tasks with ranks in the range $[1, n]$. The processes that are alive at the beginning of round r but have $age \neq 1$ are aware that $ASure \neq \emptyset$ (they receive the messages from the processes in $Asure$). Hence they perform different tasks with ranks in the range $[n + 1, 2n]$.
We now consider the case where set $ASure = \emptyset$. Note that all processes that are alive at the beginning of round r are aware that there is no process with $age = 1$ in round r. This follows easily from the fact that if there were such a process, call it p, then p would have been alive in round $r - 1$ and all processes alive at the beginning of round r would receive the message from p informing them of his age. Now, the processes that have restarted in round r ($age = 0$) perform pairwise different tasks with ranks in the range $[2, n + 1]$. It remains to consider the processes that are alive in round r and have $age > 1$. Since these processes were also alive in round $r - 1$, they know each others' ages in round r. So, although they might form inconsistent sets $Recved$ (due to failures of processes while broadcasting in the previous rounds) they will have a consistent ranking among them. So no two processes that are alive in the beginning of round r and have $age > 1$ will consider, each one, itself as the process with the smallest rank. Their inconsistency might only be on processes that have failed. As a result, the task with the smallest rank might not be performed, but in any case, the live processes will perform different tasks in the range $[1, n + 1]$ (and different from the ones with $age = 0$). This completes the proof of the claim.

Now assume, to arrive at a contradiction, that there is a round r^* in which the number of pending tasks is bigger than $\text{OPT}(\mathcal{A}, r^*) + 3n$; moreover, let r^* denote the first such round. (Here the number of task is measured at the end of each round.) Let r be the oldest round before r^* such that the number of pending tasks is at most $\text{OPT}(\mathcal{A}, r) + 2n$. It follows that in round $r + 1$ the number of pending tasks is bigger than $\text{OPT}(\mathcal{A}, r + 1) + 2n \geq 2n$ and smaller than $\text{OPT}(\mathcal{A}, r + 1) + 3n - 1$, by the fact that the algorithm performs at least one task (due to admissibility and algorithm specification) while the optimum offline solution performs at most n tasks in round $r + 1$. It follows that $r + 1 < r^*$. In the time interval $[r + 2, r^*]$, containing at least one round, the number of pending tasks at the end of each round is bigger than $2n$, and therefore by the Claim it follows that the number of tasks performed by the algorithm is the same as the number of tasks performed by the optimum offline solution. Thus, at the end of round r^* the number of pending tasks is upper bounded by

$$(\text{OPT}(\mathcal{A}, r + 1) + 3n - 1) + (\text{OPT}(\mathcal{A}, r^*) - \text{OPT}(\mathcal{A}, r + 1)) = \text{OPT}(\mathcal{A}, r^*) + 3n - 1 \ ,$$

which contradicts the assumption that this number is bigger than $\text{OPT}(\mathcal{A}, r^*) + 3n$. □

Remark 4.7. *We now argue that the lower bound stated in Theorem 3.1 is also valid for fair algorithms run against any admissible adversary, even the one restricted by t-survivability. Recall from Section 2 that a greedy offline algorithm, which in every round schedules different alive processes to different pending tasks with the largest pending time, is not only a fair algorithm (since it assures that each task is performed in a finite number of rounds) but it is also optimal with respect to the number of pending tasks. This algorithm is an example of an optimum offline solution on which the competitive results obtained in Section 3 hold. Then it follows that since the version of the problem considered in this section is harder than the one considered in the previous section (fairness requires correctness), but the optimum offline solution is the same, the lower bound stated in Theorem 3.1 for any adversary holds here as well. Hence, one may conclude that algorithm AlgCSF, as well as fair algorithms AlgCIF and AlgLIF of competitiveness OPT + $O(n)$ developed in the subsequent Sections 4.2 and 4.3 for central and local injectors, respectively, are near-optimal.*

4.2 Central Injector

Recall transformation $TranCStoCI$ from Section 3.2. It is easy to see that algorithm AlgCSF is a specialization of the generic algorithm GenCS: information x_i is the age of process i. The remaining specification of algorithm AlgCSF (along with the required data structures) is essentially the specification of the scheduling policy $\mathcal{S}$ in the setting with

central scheduler. Now, let **Algorithm AlgCIF** be the algorithm resulting by applying the transformation $TranCStoCI$ to algorithm AlgCSF (it is essentially algorithm GenCI appended with the scheduling policy of algorithm AlgCSF). Then, from Lemma 3.4, Theorem 4.5, and Theorem 4.6 we get:

Theorem 4.8. *Algorithm AlgCIF is a fair algorithm that achieves competitive pending-tasks complexity of at most OPT* $+ 3n$ *under any* 2*-survivability adversary.*

Remark 4.9. *From the observations made in Remarks 3.6 and 4.7 we conclude that algorithm AlgCIF is near-optimal.*

4.3 Local Injector

We now consider algorithm AlgLIF. This algorithm combines the mechanism deployed by algorithm AlgLI for propagating newly injected tasks with a round of delay and the scheduling policy of algorithm AlgCSF to guarantee fairness. Reliable multicast is again assumed for assuring that processes maintain consistent sets of pending tasks. See bellow a full description of algorithm AlgLIF (it is essentially a combination of the descriptions of the two above-mentioned algorithms).

Algorithm AlgLIF(i,r)

- Get specifications of newly injected tasks from local scheduler and store them in set new (remove any older information from this set).
  Receive messages sent (if any) in round $r - 1$. (From each process j process i gets the sets old_j, new_j, task id t_j, and $age_r(j)$.)
- Based on received messages, construct set $ASure$ by including all processes j with $age_r(j) = 1$. If $age = 1$, then i includes itself in the set.
- Update set old based on received messages: the new set *old* is the union of all the received sets *old* and *new* minus the tasks that have been reported in the current round as done in the previous round.
- Rank the tasks in set old *lexicographically*: first based on their pending period (older tasks have smaller rank) and then based on their task ids (incremental order).
- If the number of tasks in old is larger than $2n$ then
  - If $ASure \neq \emptyset$ then
    - If $age \neq 1$ then perform task in old with rank $n + i$.
    - Else rank processes in $ASure$ based on their ids and perform task in old with rank $rank(i)_{ASure}$.
  - Else [$ASure = \emptyset$]
    - If $age \neq 0$ then construct set $Recved$ by including all processes from whom a message was received at the beginning of the round. Process i includes itself in this set. Then rank processes in set $Recved$ lexicographically, first based on their age and then based on id (increasing order). If $rank(i)_{Recved} = 1$ then perform task in old with rank 1, otherwise perform task in old with rank $i + 1$.
    - Else perform task in old with rank $i + 1$.
- Else [old has fewer than $2n$ tasks]
  - if $old = \emptyset$ and if $new \neq \emptyset$ then rank the tasks in new incrementally based on their ids and perform task in new with the smallest rank.
  - Else perform task in old with rank 1.
- Set $age = age + 1$.
- Send sets new and old, the task id of the performed task, and value of age to all other processes.

Its competitiveness is the same as the competitiveness of AlgCSF plus an additive factor n coming from the one-round delay of the propagation of newly injected tasks. Specifically we have that:

Theorem 4.10. *Algorithm AlgLIF, assuming reliable multicast, is a fair algorithm that achieves competitive pending-tasks complexity of at most OPT* $+ 4n$ *against any* 2*-survivability adversary.*

Using similar reasoning as in the previous sections it follows that algorithm AlgLIF is near-optimal.

5 Extensions and Limitations

In this section we consider the impact of restricted communication and non-unit-length tasks on the competitiveness of the problem of performing tasks under dynamic crashes-restarts-injections patterns.

5.1 Solutions Under Restricted Communication

In view of Theorems 3.1 and 3.2, we argue that exchanging messages between processes does not help much in the setting with central scheduler, in the sense that in the best case it could slightly increase only the constant in front of the additive linear part of the formula on the number of pending tasks. In this section we study the problem of how exchanging messages may influence the correctness of solutions in more restricted settings of injectors. In particular, we show that $\Omega(n^2)$ per-round message complexity is inevitable in order to achieve correctness even in the presence of central injector. On the other hand, recall that $O(n^2)$ per-round message complexity is enough to achieve near-optimal solution in the presence of central injector: algorithm AlgCI from Section 3.2 achieves near-optimal competitiveness of at most OPT $+ 2n$ in this setting, c.f., Theorem 3.5.

Theorem 5.1. *For any algorithm and any $t \geq 1$, there is an adversarial pattern satisfying t-survivability such that the execution of the algorithm under this pattern results in $\Omega(n^2)$ per-round message complexity, even in the model with central injector.*

Proof: Fix parameter $t \geq 1$ and algorithm Alg. Consider an execution in which the adversary awakes $n/2$ processes in the beginning of round 1 (the other processes are not operational yet). One of these processes, arbitrarily selected, crashes at the end of round t, while each of the remaining awaken processes stays alive by the end of round $t+1$ (hence t-survivability is not violated). These processes carry the knowledge of pending tasks, and assume that the adversary injects $n/2+1$ new tasks in every round. Hence at the beginning of round $t+2$ there are at least 2 pending tasks. We claim that in round $t+2$ *all* the $n/2-1$ alive processes must send a message to each of the remaining $n/2+1$ processes.

Assume otherwise. Say that only one of these processes, call it i, does not send a message to each of the remaining $n/2+1$ processes. Then the adversary fails all processes but i before the sending part of round $t+2$. Admissibility is not violated since process i is alive. At the beginning of round $t+3$ the adversary crashes process i and wakes up those processes among the $n/2+1$ processes that were non-operational so far, to which process i did not send a message in round $t+2$ (these processes will be alive for the next t rounds, including round $t+3$, so neither admissibility is violated in round $t+3$, nor t-survivability). Since restarted processes are history-oblivious, the only information they get is from the central injector: new tasks injected and tasks that have been performed in the previous round. Now, since i was the only operational process and could perform at most one task in each round, there are at least three tasks that were not performed (even if crashed processes informed the central injector about the performance of their task before crashing, still at least three tasks would not be allocated to any process and hence stay unperformed), and since i did not forward the history it carried to the newly awaken processes, the information of these tasks is lost, violating correctness. □

5.2 Non-unit-length Tasks

We now turn our attention to tasks that are not necessarily of unit-length, that is, they might take longer than a round to complete. We consider a persistent setting, in which once a process commits in performing a certain task of length x, it will do so for x consecutive rounds, until the task is performed. If the process is crashed before the completion of all x rounds, then the task is not completed. We assume that processes cannot share information of partially completed tasks; the task performance is an atomic operation. In view of these assumptions, the number of pending tasks remains a sensible performance metric.

First, we consider tasks of the same length $d \geq 1$, i.e., each task takes d rounds to be performed. Consider a variation of algorithm AlgCS of Section 3.1 that uses the same scheduling policy, but once a process chooses a task to perform, it spends d consecutive rounds in doing so; call this AlgCS$_d$. We show the following:

Theorem 5.2. *Algorithm AlgCS$_d$, for uniform tasks of length d, achieves competitive pending-task complexity of at most OPT $+ 3n$ under any admissible adversarial pattern, in the setting with central scheduler.*

Proof: Assume, to arrive at a contradiction, that there is a round r^* of an execution of the algorithm under some adversarial pattern $\mathcal{A}$ in which the number of pending tasks by the end of the round is bigger than $\text{OPT}(\mathcal{A}, r^*) + 3n$. Moreover, let r^* denote the first such round. Let r be the oldest round before r^* such that the number of pending tasks by the end of this round is at most $\text{OPT}(\mathcal{A}, r) + n$.

Consider the time interval $(r^*, r]$. Since the number of pending tasks is at least n in this interval, when a process selects a task to perform, it will always be a task that has neither been performed nor is being performed by some other process (here we use the property that the central scheduler returns all tasks that have not been confirmed as performed yet). Moreover, as all tasks are of the same length d, if a process performs i tasks in the considered period of the execution of AlgCS_d, it performs at most $i + 1$ tasks in any other execution obtained under the same crash-restart-injection pattern. The additional summand 1 comes from the fact that if a process has been alive in round $r^* + 1$, it may finish its first task in this period at most $d - 1$ rounds later in the execution of AlgCS_d, comparing to the optimum solution; this may result in at most one more task being performed by the process in the optimum solution until the first crash in this period, but starting from the next restart, the timing of task completions are the same in both executions, though the actually performed tasks may be different.

Note also that each first task completed by a process in the considered period may not be unique (i.e., not attempted to be done by any other process in parallel, that is, during performance time), as it might have been selected before the interval started, and thus the number of pending tasks could have been smaller than n (i.e., not guarantying no repetition property). Hence, if the total number of tasks performed by AlgCS_d in the considered period is x, it is at most $x + 2n$ for the optimum algorithm. The first n comes from the fact that in the execution of AlgCS_d the first tasks completed by processes in the interval $(r^*, r]$ may not be distinct; the second n comes from the fact that the optimum solution may perform one more task per each process.

Therefore, the number of pending tasks $\text{OPT}(\mathcal{A}, r)$ at the end of round r in the execution of the optimum algorithm is at least $\text{OPT}(\mathcal{A}, r^*) + (y - (x + 2n))$, where y is the total number of tasks injected in the interval $(r^*, r]$. On the other hand, the number of pending tasks at the end of round r in the execution of AlgCS_d is at most

$$(\text{OPT}(\mathcal{A}, r^*) + n) + (y - x) \leq \text{OPT}(\mathcal{A}, r) + 3n \ ,$$

which is a contradiction. □

Remark 5.3. *Observe that the lower bound stated in Theorem 3.1 can be made to hold also for uniform, non-unit tasks. To see this, consider the adversarial pattern as described in the proof of Theorem 3.1, and have each round be "emulated" by d rounds. Hence, AlgCS$_d$ is near-optimal.*

We conjecture that similar techniques would allow to obtain near-optimal analysis for the other algorithms developed in this paper, in the context of the remaining two models of central and local injectors, and under the fairness requirement.

We now consider the case where tasks could be of *different* lengths. It follows that bounded competitiveness is not possible, even under restricted adversarial patterns, and even in the model with central scheduler.

Theorem 5.4. *For any algorithm, any number $n \geq 2$ of processes, any $t \geq 1$ and any upper bound $d \geq 3$ on the lengths of tasks, there is an adversarial pattern satisfying t-survivability such that the execution of the algorithm under this pattern results in unbounded competitiveness with respect to the pending task complexity, even in the model with central scheduler.*

Proof: Assume we are given an algorithm Alg. Consider any integers $n \geq 2$, $t \geq 1$ and $d \geq 3$. The adversary keeps the first process continuously alive, to guarantee admissibility, and restarts and crashes the second process in a dynamic way, to be defined later. The remaining $n - 2$ processes are kept asleep throughout the whole execution. The adversary injects only tasks of length 2 and 3 (this is enough to show the negative result).

We specify the crash/restart pattern for the second process, depending on the behavior of algorithm Alg (more precisely, the adversary emulates Alg round after round and decides when the next crash takes place in course of the simulation). In the beginning, the second process is alive. The adversary crashes the second process in the beginning of one of the three rounds: $(2j - 1) \cdot 6t + 4$ or $(2j - 1) \cdot 6t + 5$ or $(2j - 1) \cdot 6t + 6$, and restarts it in the beginning of

round $2j \cdot 6t + 1$, for any positive integer j. A decision in which round to crash — $(2j-1) \cdot 6t + 4$ or $(2j-1) \cdot 6t + 5$ or $(2j-1) \cdot 6t + 6$ — is made based on the task performed by the second process in round $(2j-1) \cdot 6t + 4$. If this task is going to be finished in round x, where x is one of $(2j-1) \cdot 6t + 4, (2j-1) \cdot 6t + 5, (2j-1) \cdot 6t + 6$, then the process is crashed in the beginning of round x. The rounds of a crash are well-defined for every j, since all tasks are of length 2 or 3.

Based on the above pattern, we define a *phase* to be the time between any two consecutive restarts of the second process. Observe that phases partition the patterns, and also the resulting executions of the algorithm, into consecutive time intervals of length $12t$ each.

The injection pattern is constructed based on the decisions of Alg and on the crashe-restart pattern defined above. There are three general rules governing task injections:

(i) Every time a process starts performing its task of length $y \in \{2, 3\}$, a new task of length y is injected into the system.

(ii) In the beginning of phase 2^j, for any integer $j \geq 0$, an additional 2^j tasks of length 2 and 2^j tasks of length 3 are injected.

The above crash/restart and injection pattern, call it $\mathcal{A}$, defines a unique execution of algorithm Alg. Define an offline algorithm OFF, not necessarily optimal, under the above crash/restart and injection pattern as follows. In each round and for each of processes one and two, it schedules a task of the same length as done in the execution of Alg, except of the two last tasks started by the second process before its crash, for each such crash. If the last task was of length 3, then algorithm OFF assigns a task of length 2 instead, while keeping the same length for the second last task. Otherwise (i.e., the length of the last task was 2), there are two sub-cases. If the second last task was of length 2, then OFF assigns a task of length 3 instead; note that OFF does not have a chance to assign any other task before the crash, as it occurs just after the assigned task of length 3. In the second sub-case, if the second last task was of length 3, then OFF assigns a task of length 2, and after that another task of length 2.

Observe that OFF does not waste any round in its execution, i.e., there is no idle round or round spent on a task that is not successfully finished; thus it is optimal for the considered pattern of crash/restarts and injections. Note also that it has always at least 1 pending task. To the contrary, Alg wastes at least one unit of work per every $2 \cdot 6t$ rounds of its execution. This means that the number of pending tasks in the considered execution of Alg is at least $OFF(\mathcal{A}, 12jt) + \lfloor j/3 \rfloor$ after round $12jt$, as it wasted at least j work units (rounds) while each task is of length at most 3; here $OFF(\mathcal{A}, 12jt)$ stands for the number of pending tasks in the execution of OFF under the considered adversarial pattern $\mathcal{A}$ in round $12jt$. Hence, from the definition of OPT it follows that the number of pending tasks of Alg is at least OPT $+ \lfloor j/3 \rfloor$. Therefore, with the growth of j, the competitiveness of Alg becomes unbounded.

It remains to be shown that OFF is well-defined, in the sense that there are pending tasks of specific lengths that can be used by OFF to alter the original assignment of Alg. This is guaranteed by the specified injection pattern. More precisely, the execution of Alg guarantees that at the end of phase j, the number of pending tasks of length 2 is at least j, and the same holds for the number of pending packets of length 3. Note that OFF, in its execution may finish at most one more task in each phase than Alg, though the length of at most one scheduled task may differ. This means that a task corresponding to one length is injected (as injections are defined based on the choices of Alg) while a task of different length is performed by OFF. However, for each phase we have two additional tasks injected (rule (ii)), of different lengths, and one of them can be used by OFF if necessary. □

6 Future Directions

Several research directions emanate from this work. An intriguing question is whether the assumption of reliable multicast, made in the setting with local injector, can be removed or replaced by a weaker but still natural constraint. We conjecture that t-survivability, for a suitable constant t, could be a good candidate for such replacement. In view of Theorem 5.1, it is challenging to find a natural restriction on the adversary such that both efficient performance and *subquadratic communication* would be achieved in the settings with injectors. For this purpose a version of the continuous gossip protocol developed in [16] could be possibly used. In view of Theorem 5.4, it would be worth

checking if randomization would help (i.e., analyzing randomized algorithms under oblivious adversaries), or whether a smoothed or average-case analysis might result in bounded competitiveness for tasks of different lengths.

Another interesting challenge is to generalize the considered task specifications to dependent tasks. Other challenging modeling extensions could involve replacing the fairness property by a more "sensitive" task latency measure, and considering energy consumption issues.

References

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

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

[3] *Amazon Elastic Compute Cloud*, `http://aws.amazon.com/ec2`

[4] H. Attiya and A. Fouren. Polynominal and adaptive long-lived $(2k-1)$-renaming. In *Proceedings of the 14th International Conference on Distributed Computing (DISC 2000)*, pages 149–163, 2000.

[5] H. Attiya, A. Fouren, and E. Gafni. An adaptive collect algorithm with applications. *Distributed Computing*, 15(2):87–96, 2002.

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

[7] Y. Bartal, A. Fiat, and Y. Rabani. Competitive algorithms for distributed data management. In *Proceedings of the* 24^{th} *ACM Symposium on Theory of Computing (STOC)*, 1992.

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

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

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

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

[12] J. Dias, E. Ogasawara, D. de Oliveira, E. Pacitti, and M. Mattoso. A Lightweight Execution Framework for Massive Independent Tasks. In *Proceedings of the* 3^{st} *IEEE Workshop on Many-Task Computing on Grids and Supercomputers*, 2010.

[13] C. Dwork, J. Halpern, and O. Waarts. Performing work efficiently in the presence of faults. *SIAM Journal on Computing*, 27(5):1457–1491, 1998.

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

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

[16] Ch. Georgiou, S. Gilbert, and D.R. Kowalski. Meeting the deadline: on the complexity of fault-tolerant continuous gossip. In Proceedings of the 29^{th} ACM Symposium on Principles of Distributed Computing (PODC 2010), pages 247–256.

[17] Ch. Georgiou, D.R. Kowalski, and A.A. Shvartsman. Efficient gossip and robust distributed computation. *Theoretical Computer Science*, 347(1):130–166, 2005.

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

[19] Ch. Georgiou, A. Russell, and A.A. Shvartsman. Work-competitive scheduling for cooperative computing with dynamic groups. *SIAM Journal on Computing*, 34(4):848–862, 2005.

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

[21] L. Hui, Y. Huashan, and L. Xiaoming. A Lightweight Execution Framework for Massive Independent Tasks. In *Proceedings of the* 1^{st} *Workshop on Many-Task Computing on Grids and Supercomputers*, 2008.

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

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

[24] G. Malewicz. A Work-Optimal Deterministic Algorithm for the Certified Write-All Problem with a Nontrivial Number of Asynchronous Processors. *SIAM Journal on Computing*, 34(4):993–1024, 2005

[25] G. Malewicz, M.H. Austern, A.J.C. Bik, J.C. Dehnert, I. Horn, N. Leiser and G. Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD conference*, pages 135–145.

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

[27] G. Malewicz, A. Russell, A. A. Shvartsman. Distributed scheduling for disconnected cooperation. *Distributed Computing*, 18(6):409–420, 2006.

[28] W. Shi and B. Hong. Resource allocation with a budget constraint for computing independent tasks in the Cloud. In Proceedings of the 2^{nd} IEEE International Conference on Cloud Computing Technology and Science, 2010

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