J. Parallel Distrib. Comput. 84 (2015) 94–107

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

On the competitiveness of scheduling dynamically injected tasks on processes prone to crashes and restarts

Chryssis Georgiou [a,*], Dariusz R. Kowalski [b]

[a] Department of Computer Science, University of Cyprus, 1678 Nicosia, Cyprus

[b] Department of Computer Science, University of Liverpool, Liverpool L69 3BX, United Kingdom

HIGHLIGHTS

- New framework for fault-tolerant job scheduling with dynamic task arrivals.
- Centralized and distributed settings considered.
- Positive results: Competitive (efficient) algorithms wrt the number of pending jobs.
- Negative results: Conditions for non-competitiveness.

ARTICLE INFO

Article history:
Received 3 October 2013
Received in revised form
14 July 2015
Accepted 17 July 2015
Available online 26 July 2015

Keywords:
Task execution
Dynamic task injection
Processor crashes and restarts
Competitive analysis
Distributed algorithms

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 tasks that are continuously and dynamically injected to the system. Tasks 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 become 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.

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., [11,12,14,21–23,28,30]) and in deploying distributed collaborative systems and applications (e.g., [2,4,15,25,27]).

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 [13,21,24]). 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 tasks can be known a priori, in todays typical Internet-based computations, such as Grid computing (e.g., [15]), Cloud services (e.g., [2]), and master–worker computing (e.g., [25,27]), tasks are generated dynamically and different tasks may become known to different processes. As such computations are becoming very popular, there is a corresponding need to develop efficient and

* Corresponding author.
E-mail addresses: chryssis@cs.ucy.ac.cy (C. Georgiou), D.Kowalski@liverpool.ac.uk (D.R. Kowalski).

http://dx.doi.org/10.1016/j.jpdc.2015.07.007
0743-7315/

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. Our investigation is based on a simple model of computation that abstracts key attributes, such as dynamic task arrivals, worst case occurrences of processes crashes and restarts, and level of information given to the processes. Our goal is to provide a rigorous analysis of the efficiency of algorithmic solutions – i.e., provide *provable* efficiency and fault-tolerance guarantees – and identify limitations even under the simple model (lower bound and impossibility results that are valid also in more complex settings). We believe that our investigation provides new insights on the complexity and fault-tolerance of dynamic task computations, on which follow up works could build on, either by enhancing the model to bypass the limitations or focus on extending our positive results to different (more complex) settings.

Basic framework: The computation is broken into synchronous rounds, in which each process is allocated 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. This abstracts the situations where tasks consume comparable resources[1] and processors are homogeneous (hence one can specify the notion of a unit-length based on the tasks' requirements and processes' capabilities). An execution of an algorithm is specified under a *crash–restart–injection* pattern, that is, a collection of crash, restart and injection events; in a crash event a process crashes, in a restart event a crashed process restarts, and in an injection event, a task in injected in the system (the task is allocated to one or many processes). 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* [32], 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; we refer to the efficiency of the offline algorithm as OPT. More precisely, we say that an algorithm has OPT + x *pending-tasks competitiveness*, if the algorithm's maximum number of pending tasks (over all rounds and executions) is greater than OPT by at most an additive real number x. (A formal definition is given in Section 2.) Observe that the comparison to OPT means in fact that we compare our algorithm's efficiency to the efficiency of *all* other possible solutions, for any crash–restart–injection pattern.

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.

[1] We refer the reader to [21] for a discussion on cooperative applications involving tasks that are independent and consume comparable resources.

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 to the system 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 [25] or Pregel [27], or as a *resource broker/scheduler* in Computational Grids such as EGEE [15].

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 to the system 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 allocates tasks to processes without giving them any global information (for example, each process may be allocated tasks that no other process in the system has been allocated, or only a subset of processes may be allocated 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* [7] is available.

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

(a) Limitations on competitiveness: We first 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 (cf. Section 3). The lower bound is proved for the model of central scheduler, but since this is the strongest information model, the result holds also for the other two weaker information models.

(b) Solutions guaranteeing correctness: Within the model of central scheduler we develop the deterministic algorithm AlgCS that does not make any use of message exchange amongst processes and achieves OPT + $2n$ pending-tasks competitiveness; in view of the lower bound above, the algorithm is optimal within a constant factor on the additive term of the competitiveness formula. 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 Section 4.

(c) 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 Section 5.

(d) Bounding communication: We show that in the models 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 Section 6.1.

(e) 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 crash–restart patterns. These results are given in Section 6.2.

The negative results of (d) and (e) give rise to interesting research questions and yield interesting future research directions. These are discussed in Section 7.

Related work. The *Do-All* problem has been studied in several models of computation, including message-passing (e.g., [13,18,21]), shared-memory (e.g., [3,8,24,26]), partitionable networks (e.g., [20]), in the absence of communication (e.g., [29]) 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* [24]. Georgiou et al. [19] 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 [19] that considers only processes crashes. Chlebus et al. in [7] 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 multicasting 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 [20], 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 [32] and it was extended for distributed algorithms in a sequence of papers by Bartal et al. [6], Awerbuch et al. [5], and Ajtai et al. [1]. Several distributed computing problems have been modeled as online problems and their competitiveness was studied. Examples include distributed data management (e.g., [6]), distributed job scheduling (e.g., [5]), distributed collect (e.g., [9]), and set-packing (e.g., [14]).

In a sequence of papers [10,11,28] 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 allocated 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 (allocated 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, cf.*, Sections 6.2 and 7, 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 [21]. 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 $r' < r$.[2] Finally an $inject(r, i, \tau)$ event specifies that process i is allocated 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 allocated 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 allocated a task τ (and this is the only process being allocated 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[3] 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 the system 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 allocated to this process; a task may be allocated to many processes in the same round.

Correctness and fairness. We consider two 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.[4] In other words, this property requires correctness, plus the guarantee that each task is eventually performed in any infinite execution of an algorithm.

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.[5] 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

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

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

[4] 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.

[5] 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.

algorithms. However a simple *greedy algorithm* scheduling different pending tasks, giving priority to the ones that have been the longest in the system, 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 $\mathcal{A}$); note that in infinite executions, this greedy algorithm is *fair*.

For the sake of more sensitive bounds on competitiveness of algorithms, we consider subclasses of adversarial patterns 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 $\text{OPT}(\mathcal{A}, r) = k$. A pattern $\mathcal{A}$ which is (k, r)-dense for some round r is called k-*dense*.

In Section 6.1 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. A lower bound on competitiveness

We begin the investigation of the complexity of our scheduling problem with a lower bound result. In particular we show that all algorithms require a linear additive factor in their competitive pending-tasks complexity even in the presence of a restricted adversary satisfying t-survivability, for any t. Our proof is developed within the model of central scheduler, therefore the derived lower bound holds for all settings considered in this work, as the central scheduler is the most restrictive one (i.e., it is the strongest information model). The result is proved for algorithms that guarantee the correctness property; since fairness is a stronger property, this lower bound trivially holds for fair algorithms as well. Finally note that the lower bound is valid even for algorithms that exchange messages (and hence information) in every round.

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 $r + 2$. Hence the result follows.

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. Hence, message exchange does not help to improve competitiveness under this adversarial pattern. □

As we will see in Sections 4, 5, and 6, we develop scheduling algorithms that achieve competitive pending-tasks complexity of $\text{OPT} + O(n)$. Therefore, with respect to the above lower bound, all these algorithms are optimal within a constant on the additive term of the competitiveness formula.

4. 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 5). We consider unit-length tasks (non-unit-length tasks are discussed in Section 6.2). 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 6.1).

4.1. Central scheduler

We begin with the strongest information model among the ones we consider in this work, that of the central scheduler. We develop a simple algorithm, called algorithm AlgCS and show that it achieves a competitive pending-tasks complexity of $\text{OPT} + 2n$. The pseudocode of the algorithm is given below, specified for a process i and a round r. Observe that the algorithm does not require sending messages between processes.

Algorithm AlgCS(i, r)

- Get set $\mathcal{P}$ of pending task specifications from the scheduler.
- Rank the task specifications in incremental order, based on the task id.
- Perform task with rank i mod $|\mathcal{P}|$.

Theorem 4.1. *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 $\text{OPT}(\mathcal{A}, r^*)$. Without loss of generality, 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*

$$\begin{aligned}&(\mathrm{OPT}(\mathcal{A}, r) + n) + \big(\mathrm{OPT}(\mathcal{A}, r^*) - \mathrm{OPT}(\mathcal{A}, r+1)\big)\\ &\quad \leq (\mathrm{OPT}(\mathcal{A}, r) + n) + \big(\mathrm{OPT}(\mathcal{A}, r^*) - (\mathrm{OPT}(\mathcal{A}, r) - (n-1))\big)\\ &\quad = \mathrm{OPT}(\mathcal{A}, r^*) + 2n - 1.\end{aligned}$$

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

One might wonder whether having the processes exchange information in every round could improve competitiveness in this setting. However, 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.

4.2. 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 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)

- (stage 1) Get set $\mathcal{P}$ of pending task specifications from the scheduler. Set $\mathcal{P}_i = \mathcal{P}$.
- (stage 2) Receive message m_j by each process j that sent a message in the previous round containing information x_j. I.e., $m_j = \langle x_j \rangle$.
- (stage 3) Based on $\mathcal{P}_i$ and each received information x_j deploy the scheduling policy $\mathcal{S}$ to perform a task.
- (stage 4) Send a message with information x_i to all other processes.

The transformation, called *Tran_CS_to_CI*, 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.

We now specify transformation *Tran_CS_to_CI* for process i and round r, when given as input algorithm GenCS(i, r).

Transformation *Tran_CS_to_CI*(i, r)

- **Replace** stage 1 with: 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.
- **Amend** stage 2: $m_j = \langle x_j, \mathcal{P}_j \rangle$ and define $R = \{j :$ received a message from j in this round$\}$.
- **Insert** a new stage: $\mathcal{P}_i = \bigcup_{j \in R \cup \{i\}} \mathcal{P}_j \cup \mathcal{N} \setminus \mathcal{D}$.
- **Keep** stage 3 the same.
- **Amend** stage 4 to also send $\mathcal{P}_i$

We now give the resulting target algorithm GenCI.

Target Algorithm GenCI(i, r)

- (stage 1) 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.
- (stage 2) Receive messages by each process j that sent a message in the previous round containing information x_j and $\mathcal{P}_j$. Let $R = \{j :$ received a message from j in this round$\}$.
- (stage 3) Set $\mathcal{P}_i = \bigcup_{j \in R \cup \{i\}} \mathcal{P}_j \cup \mathcal{N} \setminus \mathcal{D}$.
- (stage 4) Based on $\mathcal{P}_i$ and each received information x_j deploy the scheduling policy $\mathcal{S}$ to perform a task.
- (stage 5) Send a message with information x_i and $\mathcal{P}_i$ to all other processes.

Observe 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 4.2. *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 4.1. This algorithm is a specialization of algorithm GenCS where 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 $|\mathcal{P}|$. Let Algorithm AlgCI be the algorithm resulting by applying the transformation *Tran_CS_to_CI* to algorithm AlgCS. Then, from Lemma 4.2 and Theorem 4.1 we get:

Theorem 4.3. *Algorithm* AlgCI *achieves competitive pending-tasks complexity of at most OPT* $+ 2n$ *against any admissible adversary.*

4.3. 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, $\mathcal{N}$ and $\mathcal{U}$. Set $\mathcal{N}$ contains all new tasks injected to this process in this round. Set $\mathcal{U}$ 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 (i.e., they are still unperformed).

Algorithm AlgLI(i, r)

- Get specifications of newly injected tasks from local scheduler and store them in set $\mathcal{N}$ (remove any older information from this set).

Receive messages sent (if any) by other processes in round $r - 1$.

- Update set $\mathcal{U}$ based on received messages: the new set $\mathcal{U}$ is the union of all the received sets $\mathcal{U}$ and $\mathcal{N}$ 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 $\mathcal{U} \neq \emptyset$ then rank tasks in $\mathcal{U}$ incrementally based on their ids and perform task with rank i mod $|\mathcal{U}|$. Otherwise, and if $\mathcal{N}$ is not empty, then rank the tasks in $\mathcal{N}$ incrementally based on their ids and perform the task with the smallest rank.
- Send to all other processes sets $\mathcal{N}$, $\mathcal{U}$ and the task id of the performed task.

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 4.4. *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.*

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 [7]: if a process crashes while multicasting a message, then either all targeted processes (that are alive) receive the message or none does.

Lemma 4.5. *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 $\mathcal{U}$. 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 $\mathcal{N}$, $\mathcal{U}$, and the task id of the task they performed in this round (note that for each process $i \in \mathcal{L}_{r-1}$, $\mathcal{N} = I_{r-1,i}$).

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 4.4 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 $\mathcal{U}$ from the processes in $\mathcal{L}_{r-1}$ (they have the same set $\mathcal{U}$) 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 $\mathcal{U}$ and $\mathcal{N}$ and remove the tasks reported to be performed in the previous round they all form the same updated set $\mathcal{U}$. This completes the proof. □

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

Lemma 4.6. *In any execution of algorithm* AlgLI*, assuming reliable multicast, 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 4.4 we have that information on newly injected tasks is not lost; it is propagated in the system with a round of delay. 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 $\mathcal{U}$, 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 $\mathcal{U}$. 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 4.5 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 $\mathcal{N}$ 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 $\mathcal{U}$ at the beginning of round r. However,

if i also receives a report that τ was performed then it removes it from $\mathcal{U}$. 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 $\mathcal{U}$ 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 4.4 and 4.5 it follows that τ was included in the set $\mathcal{U}$ 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 $\mathcal{U}$ received by process i at the beginning of round r. And following the same reasoning as in case (a), we have 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 of algorithm AlgLI:

Theorem 4.7. *Algorithm* AlgLI*, assuming reliable multicast, is correct and achieves competitive pending-tasks complexity of at most OPT* $+ 3n$ *against any adversary.*

Proof. Correctness follows directly from Lemma 4.6. The proof of competitiveness is similar to the proof of Theorem 4.1. 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 4.4). 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 tasks in set* $\mathcal{U}$ *at the beginning of a round r, then in round r all alive processes perform pairwise different tasks.*

We now prove Claim 1. From Lemma 4.5 we have that all processes that are alive in round r have the same set $\mathcal{U}$ and per the thesis of the claim, $|\mathcal{U}| \geq n$. Regardless of whether new tasks are injected in round r, due to the scheduling policy and common knowledge of $\mathcal{U}$, 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 tasks in set* $\mathcal{U}$ *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 $\mathrm{OPT}(\mathcal{A}, r) + 3n$ 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 $\mathrm{OPT}(\mathcal{A}, r+1) + 2n$, 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 tasks in set $\mathcal{U}$ by the end of the round is bigger than $\mathrm{OPT}(\mathcal{A}, r^*) + 2n$. Moreover, let r^* denote the first such round. Let r be the oldest round before r^* such that the number of pending tasks in set $\mathcal{U}$ by the end of the round is at most $\mathrm{OPT}(\mathcal{A}, r) + n$ (it is possible that r is the first round of the execution). It follows that in round $r+1$ the number of pending tasks in set $\mathcal{U}$ is bigger than $\mathrm{OPT}(\mathcal{A}, r+1) + n \geq n$ and smaller than $\mathrm{OPT}(\mathcal{A}, r+1) + 2n - 1$, 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 tasks in set $\mathcal{U}$ 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 tasks in set $\mathcal{U}$ is upper bounded by

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

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

5. 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.

5.1. 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 5.1. *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 5.2. *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* $y \cdot (OPT(\mathcal{A}, r) + 1)$ *pending tasks at the end of round r.*

Proof. Fix a fair algorithm *Alg*. The strategy of the adversary is to repeat 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 5.2 implies that the algorithms presented in Section 4 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 5.2 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).

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 $age_{r+1}(i)$.

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

Lemma 5.3. *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 5.3 it will be performed in round $r + 1$. Hence it remains to show the following.

Lemma 5.4. *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 5.3 and 5.4 yield fairness of algorithm AlgCSF:

Theorem 5.5. *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 5.6. *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 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 $\mathrm{OPT}(\mathcal{A}, r^*) + 3n$; 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 $\mathrm{OPT}(\mathcal{A}, r)+2n$. It follows that in round $r+1$ the number of pending tasks is bigger than $\mathrm{OPT}(\mathcal{A}, r+1) + 2n \geq 2n$ and smaller than $\mathrm{OPT}(\mathcal{A}, r+1) + 3n - 1$, 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

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

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

5.2. Central injector

Recall transformation *Tran_CS_to_CI* from Section 4.2. 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 *Tran_CS_to_CI* to algorithm AlgCSF (it is essentially algorithm GenCI appended with the scheduling policy of algorithm AlgCSF). Then, from Lemma 4.2, Theorem 5.5, and Theorem 5.6 we get:

Theorem 5.7. *Algorithm* AlgCIF *is a fair algorithm that achieves competitive pending-tasks complexity of at most OPT* $+ 3n$ *under any* 2*-survivability adversary.*

5.3. 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 below a full description of algorithm AlgLIF (it is essentially a combination of the descriptions of the two above-mentioned algorithms).

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 5.8. *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.*

6. 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 crash–restart–injection patterns.

6.1. Solutions under restricted communication

In view of Theorems 3.1 and 4.1, we argue that exchanging messages between processes does not help much in the setting

Algorithm ALGLIF (i, r)

- Get specifications of newly injected tasks from local scheduler and store them in set $\mathcal{N}$ (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 $\mathcal{U}_j$, $\mathcal{N}_j$, 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 $\mathcal{U}$ based on received messages: the new set $\mathcal{U}$ is the union of all the received sets $\mathcal{U}$ and $\mathcal{N}$ minus the tasks that have been reported in the current round as done in the previous round.
- Rank the tasks in set $\mathcal{U}$ *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 $\mathcal{U}$ is larger than $2n$ then
  - If $ASure \neq \emptyset$ then
    - If $age \neq 1$ then perform task in $\mathcal{U}$ with rank $n + i$.
    - Else rank processes in *ASure* based on their ids and perform task in $\mathcal{U}$ 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 the task in $\mathcal{U}$ with rank 1, otherwise perform task in $\mathcal{U}$ with rank $i + 1$.
    - Else perform task in $\mathcal{U}$ with rank $i + 1$.
- Else [$\mathcal{U}$ has fewer than $2n$ tasks]
  - if $\mathcal{U} = \emptyset$ and if $\mathcal{N} \neq \emptyset$ then rank the tasks in $\mathcal{N}$ incrementally based on their ids and perform task in $\mathcal{N}$ with the smallest rank.
  - Else perform the task in $\mathcal{U}$ with rank 1.
- Set $age = age + 1$.
- Send sets $\mathcal{N}$ and $\mathcal{U}$, the task id of the performed task, and value of *age* to all other processes.

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 efficient solutions in the presence of central injector: algorithm ALGCI from Section 4.2 achieves competitiveness of at most OPT $+ 2n$ in this setting, cf., Theorem 4.3.

Theorem 6.1. *For any correct 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 a correct 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 (hence basic correctness holds), 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 basic correctness. □

The above lower bound can also be seen as a restriction, proved formally in the dynamic model, on the communication complexity of database transaction schedulers avoiding starvation, cf., Chapters 15.1 and 17–19 in [31].

6.2. 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 4.1 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 6.2. *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 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 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 OPT($\mathcal{A}$, r^*) at the end of round r^* in the execution of the optimum algorithm is at least 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

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

which is a contradiction. □

We note 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. This suggests that one cannot hope to obtain better competitiveness in the setting assumed in Theorem 6.2.

We conjecture that similar techniques would allow to obtain analogous analyses 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 6.3. *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 and task injection pattern for the second process, depending on the behavior of algorithm ALG and some offline algorithm OFF. The algorithm OFF does not need to be optimal for the constructed crash/restart/injection pattern, but for the purpose of our proof it is enough to show that ALG is worse than OFF by an arbitrary (unbounded) factor. In the construction, the adversary emulates ALG round after round and decides when the next crash takes place in course of the simulation, what tasks and when to inject, and how OFF schedules these tasks.

In the beginning, the second process is alive and two tasks – one of length 2 and one of length 3 – are injected into it. The pattern is partitioned into consecutive *phases* of length $t + 6$ each. The number of rounds in a phase, $t + 6$, is chosen to assure that the second process could work sufficiently long to fulfill the t-survivability requirement before it crashes (the first "t"), and in the second part (of length 6) at least two tasks could be scheduled by OFF while ALG schedules only one. Phase j lasts from round $j \cdot (t + 6) + 1$ till round $(j + 1) \cdot (t + 6)$.

We construct a suitable crash/restart and injection pattern, as well as a corresponding OFF schedule, recursively phase after phase. The goal is to satisfy the following *invariant* in the beginning of each phase $j + 1$, for $j \geq 0$:

- OFF has exactly one task of length 2 and one task of length 3 in the queue of the second process.
- ALG has at least one task of length 2 and one task of length 3 in the queue of the second process, and the total number of tasks in the queue of the second process is at least $\lfloor j/3 + 2 \rfloor$.

The invariant holds in the beginning of phase 1, by definition. Assume it holds in the beginning of phase j, we construct the crash/restart/injection pattern and OFF schedule in phase j so that the invariant also holds in the beginning of phase $j + 1$. In the beginning of the phase both processes one and two are active — the first one is always active while the adversary restarts the second one. Whenever the second process is crashed, it remains inactive by the end of the phase.

First process. The first process is never crashed, and whenever it starts performing a task of length x, x being 2 or 3, in the execution of ALG, OFF also schedules a task of length x to be performed by the first process; at the same time, the adversary injects a task of length x into the system.

Second process. We now focus on the behavior of the second process.

- Until round t of phase j: whenever ALG schedules a task of length x, x being 2 or 3, to the second process, OFF schedules a task of length x too; at the same time, the adversary injects a task of length x into the system.
- Let t^* be the first time after round t of phase j such that the second process is either idle or starts performing a task in the execution of ALG. Note that $t < t^* < t + 3$ as the longest task performed at round t of the phase is of length 3.
  - If the second process is idle or starts performing a task of length 3 in round t^* of the execution of ALG, then OFF schedules a task of length 2 for the second process, the adversary injects a task of length 2 in round $t^* + 1$ and crashes the second process at the end of round t^*. This way the second process completes one more task (of length 2) in OFF than in ALG in phase j.
  - If the second process starts performing a task of length 2 in round t^* of the execution of ALG, then OFF schedules a task of length 3 for the second process, the adversary injects a task of length 3 and crashes the second process at the end of round $t^* + 3$.

We stress here that tasks injections defined above are independent, in the sense that each injected task is uniquely triggered by exactly one of the events in the above list.

Before proving the invariant, observe that t-survivability is fulfilled as the crash of the second process occurs after round t and by round $t^* + 3 < t + 6$; this also proves that the crash occurs before the end of the phase, and the adversary can restart the second process in the beginning of the next phase.

Observe that all tasks scheduled by OFF are always successfully completed, due to the definition of crashes in a phase. Also, whenever a process (first or second) performs a task, during its execution a corresponding task of same length is injected to the system; and

since each task started to perform by the first process is immediately replaced (by injection) by another task of the same length, whenever the second process finishes a task during the execution of OFF, it can be easily argued (by induction on the completion times) that there are exactly one task of length 2 and one of length 3 in the system. This justifies the first part of the invariant.

To prove the second part of the invariant, observe that in a phase, OFF always performs more work, in terms of productive rounds, than ALG in a phase. Hence, during j consecutive phases, ALG performed at least j units of work less than OFF. Hence, at the end of phase j, ALG has at least $\lfloor j/3 \rfloor$ pending tasks more than OFF; here we used the fact that the longest task is of length 3. This proves the second part of the invariant.

It follows from the invariant that OFF has always a constant number of pending tasks, while the number of tasks queued in the execution of ALG grows unboundedly; hence the competitiveness of ALG is unbounded. □

7. Future directions

Several research directions emanate from this work. We outline a few of them below.

Message-related issues. 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 6.1, 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 [17] could be possibly used. For example, we conjecture that if the execution satisfy t-survivability and t-continuity properties, for sufficiently large constant t, bounded competitiveness could be achieved with strictly sub-quadratic message exchange in every round; we define *t-continuity* to be the property that requires each time interval of length t to have at least one continuously alive processor. In order to save on messages, a continuous gossip could be applied in place of all-to-all one-round communication, guarantying that every two processors that are continuously alive in a time interval of length at least $t/2$ exchange messages successfully. Equipped with communication tool of such property and assuming t-continuity, one could argue that it is possible to maintain a restricted number of temporary leaders who may assure lack of redundancy in case a large number of tasks are already in the system (as in such case bounded delays in information propagation become negligible).

Different task lengths. A way to overcome the impossibility stated in Theorem 6.3, as demonstrated in a recent work [16], is to consider *speed-scaling*. Processors are given a *speedup* $s \geq 1$ under which they can process a task s times faster than what is required by its specification. This can be seen as additional energy (power consumption) that processors can use to expedite computation. The challenge is to identify necessary and sufficient conditions on s under which competitiveness is possible. In [16], such conditions are identified and three different competitive deterministic algorithms are devised, each working for different bounds on s.

A different approach for tasks of different lengths would be to study whether randomization would help in achieving bounded competitiveness (when analyzing algorithms under oblivious adversaries), or whether a smoothed or average-case analysis might result in bounded competitiveness.

Task dependency. Another interesting challenge is to generalize the considered task specifications to dependent tasks. One challenge is for the processors to identify the dependency between tasks. Previous approaches assuming that the task dependency is given by a DAG do not apply in our case, as tasks are generated "on the fly" and no a priori information is known.

Alternative measures of complexity. A challenging modeling extension would involve replacing the fairness property by a more "sensitive" task latency measure. For example, one could seek solutions in which no task is executed after more than x rounds; the challenge here is to identify the conditions under such x-bounded latency solutions would be possible.

Acknowledgments

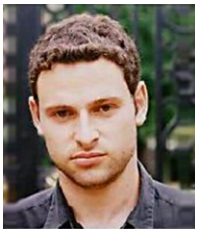
We would like to thank the anonymous referees for their fruitful feedback that has helped us to significantly improve the manuscript.

The work of first author is supported in part from research funds of the University of Cyprus (grant number UCY-ED-CG-2011). The work of second author is supported by the Engineering and Physical Sciences Research Council (grant numbers EP/G023018/1, EP/H018816/1).

References

[1] M. Ajtai, J. Aspnes, C. Dwork, O. Waarts, A theory of competitive analysis for distributed algorithms, in: Proceedings of the 35th Symposium on Foundations of Computer Science, FOCS 1994, 1994 pp. 401–411.
[2] Amazon Elastic Compute Cloud. http://aws.amazon.com/ec2.
[3] R.J. Anderson, H. Woll, Algorithms for the certified Write-All problem, SIAM J. Comput. 26 (5) (1997) 1277–1283.
[4] H. Attiya, A. Fouren, E. Gafni, An adaptive collect algorithm with applications, Distrib. Comput. 15 (2) (2002) 87–96.
[5] B. Awerbuch, S. Kutten, D. Peleg, Competitive distributed job scheduling, in: Proceedings of the 24th ACM Symposium on Theory of Computing, STOC 1992, 1992, pp. 571–580.
[6] Y. Bartal, A. Fiat, Y. Rabani, Competitive algorithms for distributed data management, in: Proceedings of the 24th ACM Symposium on Theory of Computing, STOC, 1992.
[7] B. Chlebus, R. De-Prisco, A.A. Shvartsman, Performing tasks on restartable message-passing processors, Distrib. Comput. 14 (1) (2001) 49–64.
[8] B.S. Chlebus, D.R. Kowalski, Cooperative asynchronous update of shared memory, in: Proceedings of the 37th ACM Symposium on Theory of Computing, STOC 2005, 2005, pp. 733–739.
[9] B.S. Chlebus, D.R. Kowalski, A.A. Shvartsman, Collective asynchronous reading with polylogarithmic worst-case overhead, in: Proceedings of the 36th ACM Symposium on Theory of Computing, STOC 2004, 2004, pp. 321–330.
[10] G. Cordasco, G. Malewicz, A. Rosenberg, Advances in IC-Scheduling theory: Scheduling expansive and reductive dags and scheduling dags via duality, IEEE Trans. Parallel Distrib. Syst. 18 (11) (2007) 1607–1617.
[11] G. Cordasco, G. Malewicz, A. Rosenberg, Extending IC-scheduling via the sweep algorithm, J. Parallel Distrib. Comput. 70 (3) (2010) 201–211.
[12] J. Dias, E. Ogasawara, D. de Oliveira, E. Pacitti, M. Mattoso, A lightweight execution framework for massive independent tasks, in: Proceedings of the 3st IEEE Workshop on Many-Task Computing on Grids and Supercomputers, 2010.
[13] C. Dwork, J. Halpern, O. Waarts, Performing work efficiently in the presence of faults, SIAM J. Comput. 27 (5) (1998) 1457–1491.
[14] Y. Emek, M.M. Halldorsson, Y. Mansour, B. Patt-Shamir, J. Radhakrishnan, D. Rawitz, Online set packing and competitive scheduling of multi-part tasks, in: Proceedings of the 29th ACM Symposium on Principles of Distributed Computing, PODC 2010, pp. 440–449.
[15] Enabling grids for E-sciencE (EGEE). http://www.eu-egee.org.
[16] A. Fernandez Anta, Ch. Georgiou, D. Kowalski, E. Zavou, Online parallel scheduling of non-uniform tasks: Trading failures for energy, in: Proceedings of the 19th International Symposium on Fundamentals of Computation Theory, FCT 2013, pp. 145–158.
[17] Ch. Georgiou, S. Gilbert, D.R. Kowalski, Meeting the deadline: on the complexity of fault-tolerant continuous gossip, in: Proceedings of the 29th ACM Symposium on Principles of Distributed Computing, PODC 2010, pp. 247–256.
[18] Ch. Georgiou, D.R. Kowalski, A.A. Shvartsman, Efficient gossip and robust distributed computation, Theoret. Comput. Sci. 347 (1) (2005) 130–166.
[19] Ch. Georgiou, A. Russell, A.A. Shvartsman, The complexity of synchronous iterative Do-All with crashes, Distrib. Comput. 17 (2004) 47–63.
[20] Ch. Georgiou, A. Russell, A.A. Shvartsman, Work-competitive scheduling for cooperative computing with dynamic groups, SIAM J. Comput. 34 (4) (2005) 848–862.

[21] Ch. Georgiou, A.A. Shvartsman, Do-All Computing in Distributed Systems: Cooperation in the Presence of Adversity, Springer, 2008.
[22] Ch. Georgiou, A.A. Shvartsman, Cooperative Task-Oriented Computing: Algorithms and Complexity, in: Synthesis Lectures on Distributed Computing Theory, Morgan & Claypool Publishers, 2011.
[23] L. Hui, Y. Huashan, L. Xiaoming, A lightweight execution framework for massive independent tasks, in: Proceedings of the 1st Workshop on Many-Task Computing on Grids and Supercomputers, 2008.
[24] P.C. Kanellakis, A.A. Shvartsman, Fault-Tolerant Parallel Computation, Kluwer Academic Publishers, 1997.
[25] E. Korpela, D. Werthimer, D. Anderson, J. Cobb, M. Lebofsky, SETI@home: Massively distributed computing for SETI, Comput. Sci. Eng. 3 (1) (2001) 78–83.
[26] G. Malewicz, A work-optimal deterministic algorithm for the certified Write-All problem with a nontrivial number of asynchronous processors, SIAM J. Comput. 34 (4) (2005) 993–1024.
[27] G. Malewicz, M.H. Austern, A.J.C. Bik, J.C. Dehnert, I. Horn, N. Leiser, G. Czajkowski, Pregel: A system for large-scale graph processing, in: Proceedings of the 2010 ACM SIGMOD Conference, pp. 135–145.
[28] G. Malewicz, A.L. Rosenberg, M. Yurkewych, Toward a theory for scheduling dags in Internet-based computing, IEEE Trans. Comput. 55 (6) (2006) 757–768.
[29] G. Malewicz, A. Russell, A.A. Shvartsman, Distributed scheduling for disconnected cooperation, Distrib. Comput. 18 (6) (2006) 409–420.
[30] W. Shi, B. Hong, Resource allocation with a budget constraint for computing independent tasks in the Cloud, in: Proceedings of the 2nd IEEE International Conference on Cloud Computing Technology and Science, 2010.
[31] A. Silberschatz, H.F. Korth, S. Sudarshan, Database System Concepts, McGraw-Hill, 2011.
[32] D. Sleator, R. Tarjan, Amortized efficiency of list update and paging rules, Commun. ACM 28 (2) (1985) 202–208.



Chryssis Georgiou is an Associate Professor in the Department of Computer Science at the University of Cyprus. He holds a Ph.D. (December 2003) and M.Sc. (May 2002) in Computer Science & Engineering from the University of Connecticut and a B.Sc. (June 1998) in Mathematics from the University of Cyprus. His research interests span the Theory and Practice of Fault-tolerant Distributed and Parallel Computing with a focus on Algorithms and Complexity. He has published in top journals and conference proceedings in his area of study and he has co-authored two books on Robust Distributed Cooperative Computing. He served on Program Committees of top conferences in Distributed Computing and on the Steering Committee (2008–2010, 2010–2012) of the International Symposium on Distributed Computing (DISC). Currently he serves on the Steering Committee (as Treasurer) of the ACM Symposium on Principles of Distributed Computing (PODC).

Dariusz R. Kowalski is a Professor in the Department of Computer Science at the University of Liverpool, United Kingdom. He received his Ph.D. in computer science in 2001 and M.Sc. in mathematics in 1996, both from the Warsaw University, Poland. His research interests focus on algorithmic aspects of computation and communication, especially in the areas of distributed computing, network protocols and fault-tolerance.