Coordinated Cooperative Task Computing Using Crash-Prone Processors with Unreliable Multicast[☆]

Seda Davtyan[a], Roberto De Prisco[b], Chryssis Georgiou[c,*], Theophanis Hadjistasi[d], Alexander A. Schwarzmann[d]

[a] *JW Player Inc, New York, NY 10016, USA.*
[b] *Università di Salerno, 84084 Fisciano (SA), Italy.*
[c] *University of Cyprus, 1678 Nicosia, Cyprus*
[d] *University of Connecticut, Storrs, CT 06269, USA*

Abstract

This paper presents a new message-passing algorithm, called *Do-UM*, for distributed cooperative task computing in synchronous settings where processors may crash, and where any multicasts (or broadcasts) performed by crashing processors are unreliable. We specify the algorithm, prove its correctness and analyze its complexity. We show that its worst case available processor steps is $S = \Theta\left(t + n\frac{\log n}{\log\log n} + f(n-f)\right)$ and that the number of messages sent is less than $n\left(2t + \frac{nf}{2}\right)$, where n is the number of processors, t is the number of tasks to be executed and f is the number of failures. To assess the performance of the algorithm in practical scenarios, we perform an experimental evaluation on a planetary-scale distributed platform. This also allows us to compare our algorithm with the currently best algorithm that is, however, explicitly designed to use reliable multicast; the results suggest that our algorithm does not lose much efficiency in order to cope with unreliable multicast.

Keywords: Task computing, Fault-tolerant distributed algorithms, Crash faults, Unreliable multicast.

[☆] A preliminary version of this work appears in [1].

[*] Corresponding author – chryssis@cs.ucy.ac.cy, +35722892745 (tel), +35722892701 (fax)

Email addresses: `seda@jwplayer.com` (Seda Davtyan), `robdep@unisa.it` (Roberto De Prisco), `chryssis@cs.ucy.ac.cy` (Chryssis Georgiou), `theo@uconn.edu` (Theophanis Hadjistasi), `aas@uconn.edu` (Alexander A. Schwarzmann)

1. Introduction

With the end of Moore's Law in sight, parallelism became the main means for speeding up computationally-intensive applications, especially in the cases where large collections of tasks need to be performed. Network supercomputing—taking advantage of very large numbers of computers in distributed environments—is an effective approach to massive parallelism that harnesses the processing power inherent in large networked settings. In such settings, processor failures are no longer an exception, but the norm. Any algorithm designed for realistic settings must be able to deal with failures.

Network supercomputing enables harnessing the immense computational power of the global Internet platform. A typical Internet supercomputer consists of a master computer or server and a large number of computers called workers, performing computation tasks on behalf of the master, cf. [2, 3]. Despite the simplicity and benefits of a single master approach, as the scale of such computing environments grows, it becomes unrealistic to assume the existence of an infallible master that is able to coordinate the activities of multitudes of workers. Furthermore, large-scale distributed systems are inherently dynamic and are subject to perturbations, such as failures of computers and network links, thus it is also necessary to consider fully distributed peer-to-peer solutions. In the study of cooperative algorithms, the standard abstract problem is called *Do-All* [4], and is defined as follows.

Do-All: n processors must cooperatively perform t tasks in the presence of adversity.

Simple but unrealistic solutions of the problem make use of a master processor which is not subject to failures and thus can easily coordinate the execution of the tasks. To remove the troublesome assumption of having an infallible master, coordinator-based solutions have been proposed: the approach is to use more than one coordinator to manage the computation, and if coordinators fail, they are dynamically replaced by other coordinators. Coordinator algorithms are usually simpler and more practical than other peer-to-peer algorithms. In

particular, single-coordinator algorithms can be very efficient when failures are infrequent, and they have substantial advantages over master-worker algorithms as they allow any processor to act as a coordinator, thus eliminating a single point of failure. However, such single coordinator solutions become very inefficient in adversarial settings where there are stretches of time during which each appointed coordinator crashes without accomplishing the needed coordination [5].

Thus, it is advantageous to explore solutions that allow multiple concurrent coordinators. Using multiple coordinators raises the new challenge of dealing with potential inconsistency in workers' states in the cases when, due to failures, multiple coordinators provide different information to the workers. To solve this problem, some algorithms resorted to using reliable broadcast, where the messages are delivered either to all destinations or to none if the sender crashes during the broadcast. With reliable broadcast/multicast, whenever coordinators share information with each other, it is guaranteed that all non-crashed coordinators will receive the same messages, leading to a consistent state.

In synchronous deterministic settings, combining reliable broadcast with coordinator-based approach allows one to construct very efficient algorithms, such as Algorithm *AN*, presented in [6]. However, reliable broadcast is a very strong assumption. Thus it is interesting to explore multiple-coordinator solutions that do not rely on reliable broadcast. Interestingly, algorithm AN may become very inefficient if used with unreliable broadcast because the participants may be fragmented into two or more groups, with each group believing that all other processors crashed.

Contributions. In this paper we aim to advance the state of the art in this direction, viz., to design algorithms for synchronous settings that use multiple coordinators to achieve good performance in the presence of processor crashes, while using only unreliable broadcast. We aim to develop efficient point-to-point message-passing algorithms for the *Do-All* problem that are simple and easy to implement. We note that there is a simple brute-force approach, where

all processors always broadcast all of their knowledge to all other processors (as suggested in [6]). Such an *all-to-all* solution is very efficient in terms of work, but its message complexity is prohibitive. Thus we aim for a more balanced algorithm that trades work for better communication efficiency. As our point of departure, we use the multi-coordinator approach introduced in algorithm *AN* [6] (see Section 2 on related work for more details about algorithm *AN*). In this paper, we present an iterative algorithm that is simpler than algorithm *AN* in that each iteration consists of two send-receive-compute stages as compared to three similar stages in algorithm *AN*. More importantly, the new algorithm does not assume reliable multicast, and it is not subject to the partitioning problem in algorithm *AN*. In more detail our contributions are as follows.

1. We present a new algorithm, called algorithm *Do-UM*, that is designed to work with both unreliable and reliable multicast and that tolerates up to $n-1$ crashes. The algorithm solves *Do-All* for n processors and t tasks, where $n \leq t$. When using reliable multicast, the algorithm has the same work and message complexity bounds as algorithm *AN* (this can be shown using the same analysis as in [6]).

2. We prove that algorithm *Do-UM* solves the *Do-All* problem when the multicast is unreliable. That is, we show that if the algorithm terminates, then all tasks have been performed. We also show that no correct processor is ever considered faulty by any other processor (thus the algorithm does not suffer from the partitions of correct processors that are possible in algorithm *AN* [7]).

3. We prove that the algorithm has the worst case available processor steps $S = \Theta\left(t + n\frac{\log n}{\log\log n} + f(n-f)\right)$ and that it uses at most $n\left(2t + \frac{nf}{2}\right)$ messages, where f is the number of crashes.

Thus from a theoretical point of view, algorithm *Do-UM* has an available processor steps complexity similar to that of algorithm *AN*. However algorithm *Do-UM* does not require reliable multicast.

4. To support the theoretical analysis, we also implemented algorithms *Do-UM* and *AN* using YALPS [8] and evaluated them experimentally over PlanetLab [9].

The experiments assert the performance of the algorithms in real-life scenarios. Since algorithm *AN* has been designed explicitly for settings with reliable multicast, we run the tests for *AN* using reliable multicast. Instead, for algorithm *Do-UM* the tests have been run with unreliable multicast. The results of the
95 tests show that *Do-UM* does not lose too much efficiency in order to be able to cope with unreliable multicast (we remark that the performance of algorithm *AN* is measured under reliable multicast).

As we noted earlier, in settings with reliable multicast, the efficiency of algorithms *Do-UM* and *AN* is equivalent. We also note that algorithm *AN* remains
100 correct when multicast/broadcast is unreliable: this is so for the very simple reason that each active processor, regardless of any coordination or lack of it, will ultimately perform all undone tasks by itself, thus solving the problem. However the efficiency analysis of *AN* does not apply to the setting with unreliable broadcast, and the performance of the algorithm can be rather poor. The
105 algorithm operates under the assumption that all live processors receive consistent information. In the event that this does not happen (e.g., when a processor crashes in the middle of a broadcast, and broadcast is not reliable, some processors may receive the message and some not), then the set of processors may partition into several groups where different knowledge about the number of live
110 processors and pending tasks, leading to inefficient load balancing and the high resulting task execution redundancy.

Thus, if we consider settings with unreliable broadcast, *AN* becomes a poor choice, while algorithm *Do-UM*, as the theoretical analysis and the results of the experiments show, exhibits a behaviour that is only slightly worse than
115 that of the reliable broadcast case. This demonstrates the benefit and value of algorithm *Do-UM*.

Tables 1 and 2 summarize the comparison of algorithms *Do-UM* and *AN*.

Document structure. Section 2 presents related work. Section 3 gives the model of computation, definitions, and measures of efficiency. We present and
120 analyse algorithm *Do-UM* in Sections 4 and 5. Section 6 discusses the experi-

Algorithm	**Communication**	**Theoretical Performance**
AN	Designed for reliable broadcast	$S = O((t + n\log n/\log\log n)\log f)$ $M = O(t + n\log n/\log\log n + fn)$
Do-UM	Works with unreliable broadcast	$S = \Theta\left(t + n\frac{\log n}{\log\log n} + f(n-f)\right)$ $M \leq n\left(2t + \frac{nf}{2}\right)$

Table 1: Theoretical comparison of *AN* and *Do-UM*

Algorithm	**Experimental Performance**	
	Reliable broadcast	Unreliable broadcast
AN	good	poor
Do-UM	comparable to *AN*	much better than *AN*

Table 2: Experimental comparison of *AN* and *Do-UM* (details in Section 6)

mental evaluation of algorithms *Do-UM* and *AN*. We conclude in Section 7.

2. Prior and Related Work

The *Do-All* problem has been studied in a variety of settings, e.g., in *shared-memory* models [10, 11, 12, 13, 14, 15], in *message-passing* models [4, 5, 6, 16, 17, 18, 19, 20, 21, 22] and in *partitionable networks* [23, 24]. Kanellakis and Shvartsman [10], who initially formulated the *Do-All* problem, introduce the *available processor steps* work complexity measure S and provide a lower bound of $S = \Omega(t + n\log n/\log\log n)$ for synchronous crash-prone processors, and this bound holds both for the shared-memory model and for message-passing. A comprehensive treatment of the *Do-All* problem in the message-passing setting is given in [4]; basic techniques used in solving *Do-All* can be found in [7].

Dwork, Halpern, and Waarts [16] consider the *Do-All* problem in the message-passing model and develop several deterministic algorithms for synchronous crash-prone processors. They evaluate the efficiency of their solutions using the *task-oriented* complexity measure that accounts only for task executions. Most work in the message-passing model focus on synchronous models (for an example of an asynchronous setting see [25]). The work in [5] presents

an algorithm for *Do-All* for the synchronous setting with processor crashes using the available processor steps work complexity. In particular, it provides a deterministic algorithm that has work $O(t + (f + 1)n)$ and message complexity (total number of point-to-point messages sent) $O((f + 1)n)$, where f is the number of processor crashes. The algorithm deploys a single-coordinator approach: At each step all processors have a consistent (over)estimate of the set of all the available processors (using checkpoints). One processor is designated to be the coordinator. The coordinator allocates the undone tasks according to a certain load balancing rule and waits for notifications of the tasks which have been performed. The coordinator changes over time (due to coordinator crashes). To avoid a quadratic upper bound, substantial processor slackness is assumed ($n \ll t$). Paper [5] also shows a work lower bound of $\Omega(t + (f + 1)n)$ for any algorithm using the stage-checkpoint strategy; this bound is quadratic in n for f comparable with n. Moreover, any protocol that has at most one active coordinator is bound to have work $\Omega(t + (f + 1)n)$.

The work in [26] provides a *Do-All* algorithm that beats the lower bound shown in [5]; this work does not assume reliable multicast, and instead of using coordinators it pursues a gossip-based coordination technique in conjunction with scheduling based on permutations satisfying certain properties and expander graphs. The results in [27] show how to construct the needed permutations efficiently, yet the algorithm in [26] remains complex and hard to implement.

The work in [6] presents algorithm *AN* whose work also beats the lower bound of [5] by using multiple coordinators. In order to achieve this, the algorithm, unlike other solutions, uses *reliable multicast*, where if a processor crashes while multicasting a message, this message is either received by all targeted processors or by none. The algorithm operates in iterations. Initially there is a single coordinator. If it fails in some iteration, then two processors become coordinators in the next iteration. If they both fail, then four processors act as coordinators, and so on. If in a given iteration at least one of the coordinators survives the iteration, then the next iteration has one coordinator.

Algorithm *AN* achieves work $O((t + n \log n/\log\log n) \log f)$ and message complexity $O(t + n \log n/\log\log n + fn)$, where $f < n$ is a bound on the number of processor crashes. The reliable multicast assumption is essential for maintaining consistent views of the processors in the presence of multiple coordinators. If this assumption is removed, the algorithm is still able to solve the problem, but some executions of the algorithm may be very inefficient because the set of processors may partition into several groups (see [7]).

The *Do-All* problem has also been considered in other models of failure, in particular in models with Byzantine failures [28]. The Byzantine case is quite challenging due to the more virulent adversarial setting where tasks may be performed incorrectly by faulty processors. Thus the performance of algorithms that tolerate Byzantine failures is worse compared to the algorithms that deal with crash failures. To deal with Byzantine failures, models have been proposed that allow processors within a single constant time stage to verify that a certain number of tasks were performed correctly.

Finally, we remark that in [1] we have provided preliminary results which have been extended in this paper. More specifically the differences between this paper and [1] can be summarized as follows.

- In this paper we provide a theoretical analysis of the work and message complexities of *Do-UM*. In [1] only the correctness of the algorithm is proven; the complexity analysis is mentioned as future work. The analysis provided in this paper first presents an upper bound on the work (available processor steps) complexity of *Do-UM*, and then a lower bound showing the tightness ($\Theta()$) of the upper bound analysis. Then, the message complexity of the algorithm is shown. The analysis increases significantly the understanding of the algorithm, especially on how the algorithm is handling subtle issues. Furthermore, the theoretical analysis allows a precise comparison of the efficiency of *Do-UM* with that of *AN*, the currently best algorithm that requires reliable multicast (the new algorithm does not require reliable multicast). Also, the formal analysis shows that, in

200 the presence of reliable multicast, the new algorithm has the same (asymptotic) complexity as algorithm *AN*.

- In [1] an experimental evaluation via simulation is presented. Instead, in this paper, we have implemented (using YALPS) *Do-UM* (and algorithm *AN*) and run real-life experiments on PlanetLab (an adverse planetary-scale distributed platform). This not only demonstrates the practicality of *Do-UM* but it also enables us to compare its performance with *AN* under realistic scenarios. This evaluation has resulted in interesting observations (see Section 5).

3. Model of Computation and Definitions

Processors and Tasks. We consider a set of n processors $P = \{p_1, \ldots, p_n\}$, each with a unique processor identifier (pid). For simplicity we will let the pid of processor p_i to be i, for $i \in [n]$. The processors must execute a set of t tasks $T = \{\tau_1, \ldots, \tau_t\}$. Each task has a unique identifier and we let the identifier of task τ_i to be i, for $i \in [t]$. Processors obtain tasks from some repository (or processors initially know all tasks). The tasks are (a) similar, meaning that any task can be done in constant time by any processor, (b) independent, meaning that each task can be performed independently of other tasks, and (c) idempotent, meaning that the tasks admit at-least-once semantics and can be performed concurrently. Several applications involving tasks with such properties are discussed in [4].

Computation. The system is synchronous and computation proceeds in *steps*, where each step has a fixed and known duration. In any step a processor can either send or receive messages, or can perform some local computation. We define a computation *stage* to consist of a *Send* step, a *Receive* step, and a *Compute* step.

Communication. Processors communicate by exchanging messages. We assume that any processor can send messages to every other processor, that is, the

underlying communication network is fully connected. Processors can communicate by means of point-to-point messages and multicasts, where a message to each multicast destination is treated as a point-to-point message. Point-to-point messages are not lost or corrupted, however multicasts are not reliable in the sense that if a processor crashes during a multicast, then some arbitrary subset of the target processors receives the message. We assume that there is a known upper bound on message delivery time and that if a message sent to a processor in a *Send* step, then it is delivered in the subsequent *Receive* step (provided the receiving processor does not crash). Note that in any step a processor may receive up to n messages, thus we assume that the time needed to process a received message is insignificant compared to the duration of the step. We let the set M stand for all possible messages.

Model of failures. Processors may crash (stop) at any point in the execution, subject only to the constraint that at least one processor does not crash. In particular, a processor can crash during a Send step. In this case it is possible that the messages sent are received by some recipients and not by others. Once crashed, a processor does not recover. In stating our results we use f to stand for the allowable number of crash failures.

The problem. The *Do-All* problem requires the execution of the tasks T in the distributed system consisting of the processors P, where the processors are subject to the above failure model.

Measures of efficiency. *Do-All* algorithms are evaluated in terms of *work complexity* and *communication complexity.* Work complexity is given as the total number of steps, i.e., we use available processor steps complexity [10]. In our algorithm each iteration consists of a constant number of steps and each correct processor performs one task in each iteration. Thus, asymptotically, it is sufficient to assess the total number of times each task is executed (a task may be performed more than once). Message complexity is given by the total number of point-to-point messages sent during an execution, where a multicast contributes as many messages as there are destinations.

4. Algorithm Description

We now present algorithm *Do-UM*. All processors act as *workers* and perform tasks according to a load balancing rule; additionally, some workers also act as *coordinators* using a multi-coordinator approach. The algorithm proceeds in iterations, each consisting of two stages, a Collect stage and a Disseminate stage. The algorithm uses two types of messages: the `report` message and the `summary` message. Each message contains three values: a set of done task ids, a set of failed processor ids, and the id of the sender.

In the Collect stage each processor sends a `report` message to every coordinator. Every processor that receives a `report` message in this stage updates its knowledge according to the information contained in the messages. In the subsequent Disseminate stage, coordinators send `summary` messages to all processors. A processor p acts as a coordinator when it either believes to be a coordinator, or if it receives a `report` message in the preceding Collect stage from some other processor q that considers p to be a coordinator. Failure of processors, and in particular failure of coordinators, can prevent global progress. In order to cope with coordinator failures the algorithm uses a martingale strategy: if a processor suspects that all coordinators crashed in a given iteration, it doubles the number of coordinators for the next iteration. In order to handle multiple coordinators we will use a layered structure. Each level of the structure is a "layer" of coordinators to be used in a given iteration. The first layer consists of a single processor, and each following layer has twice as many processors as the preceding layer.

The above approach is similar to that of algorithm *AN* [6]. However, there are fundamental differences. The most important is that algorithm *AN* makes a *very strong assumption of reliable multicast*, making it possible for the local knowledge of processors to be consistent. This is no longer true with unreliable multicast that might cause processors to have different views of the current status of the system (i.e., performed tasks and crashed processors). Thus, while in algorithm *AN* all workers agree on the set of coordinators, in algorithm *Do-*

UM two processors p and q may have inconsistent views of coordinators. E.g., q may be a coordinator in p's view, but not in its own view, or q may be a coordinator in its own view, but only a worker in p's view. To deal with
290 such inconsistencies in local knowledge, a processor has to be able to react to unexpected messages. Accordingly, in algorithm *Do-UM* if a worker receives an unexpected `report` message, that is, the message addressed to a coordinator, it acts as a coordinator in the second stage of the iteration. Note that the case of "unexpected" `summary` messages never arises because every processor
295 is ready to receive summary messages in each iteration from processors acting as coordinators (if any). If a processor receives at least one summary message in an iteration, it considers the iteration to be "attended" by (at least one) coordinator; otherwise it considers the iteration to be "unattended."

The main state variables at each processor are D, recording the set of ids of
300 the tasks that the processor knows to be complete, and F, the set of pids of the processors that the processor knows as crashed. Sets D and F are communicated by the processors in `report` and `summary` messages. The algorithm terminates when $D = T$, i.e., all tasks are done. Next we present the data structure used to implement the martingale strategy for selecting coordinators, then we give
305 and explain the code for the algorithm.

Layered coordinator structure. Each processor computes locally the set L of correct (non-crashed) processors ids as $P - F$. The pids in set L are listed in the ascending order, and L is interpreted as a structure of $h = 1 + \lceil \log_2 |L| \rceil$ layers. If $L = \langle q_1, q_2, ..., q_k \rangle$, for $k = |L|$, at processor p, it is interpreted
310 by p as follows. The first layer, denoted $L(0)$, is the set $\{q_1\}$, consisting of one processor. Each layer $L(\ell)$ for the next $h - 2$ layers, with $\ell \in [h - 2]$, is defined to be $\{q_{2^\ell}, \ldots, q_{2^{\ell+1}-1}\}$. The lowest layer $L(\ell)$ for $\ell = h - 1$ consists of the remaining pids, being $\{q_{2^\ell}, \ldots, q_k\}$. Thus the number of processors in any $L(\ell)$ is 2^ℓ, except possibly for the lowest layer that may contain fewer pids if
315 $k < 2^h - 1$.

Initially, any processor p has $\ell = 0$ and it considers the processor in $L(0)$ to be

the coordinator. In each iteration where p does not hear from any coordinators, it increments ℓ, and considers processors in $L(\ell)$ as the next coordinators. Thus, following each iteration unattended by coordinators, the number of coordinators is doubled (until all processors in the lowest layer act as coordinators). Following each attended iteration, ℓ is reset to 0, leaving one processor as the coordinator.

Example.

Suppose that we have a system of $n = 13$ processors. Initially processor p assumes that all the processors are alive. The layered structure $L = \langle q_1, q_2, \ldots, q_{13}\rangle$ is as below:

Layered coordinator structure L

Layer 0						1				
Layer 1			2						3	
Layer 2		4		5				6		7
Layer 3	8	9		10	11		12	13		

Layer $L(0)$ consists of processor 1, layer $L(1)$ consists of processors 2 and 3, etc. At some point later in the computation, processor p may learn that processors $1, 2, 5$, and 12 crashed, setting F accordingly. When the local view is updated taking into account F we have $L = \langle p_3, p_4, p_6, \ldots, p_{11}, p_{13}\rangle$ and the corresponding structure becomes:

Updated layered coordinator structure L at p

Layer 0					3			
Layer 1			4				6	
Layer 2		7		8		9		10
Layer 3	11	13						

Algorithm details. We now describe the algorithm in greater detail. The algorithm proceeds in iterations and each iteration consists of two stages. Recall that a stage consists of a Send step, a Receive step, and a Compute step. The pseudocode for the algorithm is shown in Figure 1.

The `report` and the `summary` messages contain triples (D, F, i) consisting of a set D of task identifiers, a set F of processor identifiers, and the pid i of the sender. For any message m we denote by $m.D$ the set D of (done) task

Algorithm *Do-UM* for processor p_i

1: **external** T, P
2: $D : 2^T$ **init** $\emptyset$ /* *set of done tasks* */
3: $F : 2^P$ **init** $\emptyset$ /* *set of crashed processors* */
4: $L : 2^P$ **init** P /* *coordinator layered structure* */
5: $\ell : \mathbb{N}^{\geq 0}$ **init** 0 /* *current level in* L */
6: $C : 2^P$ **init** $\emptyset$ /* *last active coordinators* */
7: $R : 2^M$ **init** $\emptyset$ /* *set of received report messages* */
8: $S : 2^M$ **init** $\emptyset$ /* *set of received summary messages* */

9: **repeat**
10:   COLLECT STAGE
11:     *Send:*
12:       send `report(`D, F, i`)` to all $j \in L(\ell)$
13:     *Receive:*
14:       $R :=$ set of received `report` messages
15:     *Compute:*
16:       $D := D \cup \bigcup_{m \in R} m.D$
17:       $F := F \cup \bigcup_{m \in R} m.F$
18:   DISSEMINATE STAGE
19:     *Send:*
20:       If $i \in L(\ell)$ or $R \neq \emptyset$ then
21:         send `summary(`D, F, i`)` to all $j \in P - F$
22:     *Receive:*
23:       $S :=$ the set of received `summary` messages
24:       $C := \{m.pid \mid m \in S\}$ /* *Acting coordinators* */
25:     *Compute:*
26:       $D := D \cup \bigcup_{m \in S} m.D$
27:       $F := F \cup \bigcup_{m \in S} m.F$
28:       $F := F \cup L(\ell) \setminus C$
29:       If $(C \neq \emptyset)$ then /* *Coordinators attended* */
30:         $L := P - F$
31:         $\ell = 0$
32:       Else /* *No coordinator attended* */
33:         $\ell := \ell + 1$
34:       Let τ be $Bal(T - D, P - F, i)$
35:       Perform task τ
36:       $D := D \cup \{\tau\}$
37: **until** $D = T$

Figure 1: Algorithm *Do-UM* at processor $p_i \in P$.

identifiers contained in m, by $m.F$ the set F of (crashed) processor identifiers in m and by $m.pid$ the pid i contained in m, that is the sender of the message.

Collect stage. In the Send step (line 12) each processor sends a `report` message to the coordinators. In the Receive step (line 14) processors receive the `report` messages sent in the Send step. (The messages sent in the very first stage do not contain useful information, although, as a local optimization, we could let the coordinator detect crashes of processors that fail to send a message; we choose to keep the code simple.) In the Compute step (lines 16-17), using the information received in the messages each processor updates its D and F.

Disseminate stage. In the Send step (lines 20-21), any processor that either considers itself a coordinator, or receives a `report` message in the Collect stage, sends `summary` messages to all non-crashed processors. In the Receive step (lines 23-24) processors receive the `summary` messages sent in the Send step. In the Compute step (lines 26-37), sets D and F are updated using the information received in the messages. The processors also update the layered coordinator structure L by removing coordinators that failed to send the `summary` message. If coordinators were silent (i.e., crashed) and the iteration is unattended, processors follow the martingale strategy, incrementing ℓ, so that in the next iteration the number of coordinators is doubled.

At this point each processor uses the load balancing rule Bal, performs a task and records this in D.

Load balancing rule Bal. We consider the following simple load balancing rule $Bal(T - D, P - F, i)$ for each processor p: the processor ranks the tasks in $T - D$ with respect to task ids and ranks the processors in $P - F$ with respect to processor ids. Say that processor p has rank r in $P - F$. Then p chooses the task with rank $r \bmod |T - D|$ to perform.

5. Analysis of Algorithm *Do-UM*

In this section we analyse the algorithm. We start, in Section 5.1, by proving that the algorithm is correct, that is it executes all the tasks. Then in Section 5.2

365 we assess the performance of the algorithm by giving upper and lower bounds on its running time.

5.1. Correctness

We now show that in any execution with up to $n-1$ crashes algorithm *Do-UM* solves the *Do-All* problem, and that in each execution a processor considers

370 another processor as crashed only if that processor actually crashed (thus the set of active processors never partitions).

We first show that no task is ever considered to be performed unless the task has indeed been performed. Note that a task is considered performed if at least one processor completes its execution; if a processor crashes during a task

375 execution, then the task remains not done. (Note also that if a processor crashes after executing a task and before it communicates this fact to other processors, then the task is still considered not done.)

We number the iterations of the algorithm in an execution consecutively, starting with 1. In the following, we denote by D_i^k and by F_i^k, the value of,

380 respectively, set D and set F at processor p_i at the *end* of iteration k. We let D_i^0 and by F_i^0 stand for the initial values of D_i and F_i respectively. When the iteration number is implicit, we use D_i and F_i to denote the sets D and F at processor p_i. We use the notation m^k to denote a message m sent in iteration k.

385 Intuitively, if a new task τ is included in D_i^k, then either processor i executes the task in iteration k, or processor i learns from some other processor about the execution of task τ. In the first case it is obvious that the task is performed. In the second case we can get to the same conclusion by an inductive argument (on the number of iterations).

390 **Lemma 5.1.** *In any execution of algorithm Do-UM, if a task τ is in D_i^1 for any processor p_i, then task τ has been executed by processor p_i.*

Proof. Prior to iteration 1 we have $D_i = \emptyset$ for any i, thus $m.D = \emptyset$ in all report messages in iteration 1, and consequently the same holds for all summary

messages, since $D_i = \emptyset$ at the end of the Collect stage. Hence the only way for a task τ to be added to D_i^1 is for τ to be executed by processor i and added in line 36. □

Lemma 5.2. *In any execution of algorithm Do-UM, if a task τ is in D_i^k for some processor p_i in iteration $k \geq 2$, then either task τ is executed by processor p_i in iteration k, or $t \in D_j^{k-1}$ for some processor p_j.*

Proof. Consider iteration $k \geq 2$ of any execution of the algorithm. Let τ be a task in D_i^k. If $\tau \in D_i^{k-1}$ we are done. Otherwise processor i adds τ to D_i in lines 16, 26, or 36.

Case 1: τ is added in line 16 or 26. In this case it means that $\tau \in m.D$ for some message m sent in iteration k. But this implies that $t \in D_j^{k-1}$ for some processor j.

Case 2: τ is added in line 36. In this case task τ is executed by p_i. □

This leads to the following lemma.

Lemma 5.3. *In any execution of algorithm Do-UM, if a task τ is in D_i^k for some processor p_i, then task τ is executed at least once in iterations $1, 2, ..., k$.*

Proof. By induction on the number of iterations using Lemma 5.1 for the base case and Lemma 5.2 in the inductive step. □

We next show that a processor is never wrongfully considered to be faulty: if some processor i considers another processor j as crashed, then it is indeed the case that j crashed. We start by proving the following.

Lemma 5.4. *In any execution of algorithm Do-UM, if processor j is in F_i^1 of some processor i, then $j = 1$ and it crashed in iteration 1.*

Proof. By the code, processor j can be added by processor i to F_i in lines 17, 27, or 28. In the Collect stage of iteration 1, $F_p = \emptyset$ for all p, thus $m.F = \emptyset$ for any message m. The only possibility for i to add j to F_i is in line 28. The only

420 processor that sends summary messages in iteration 1 is processor 1, so it must be that $j = 1$. The only way for j to be added to F_i is for j to not be in C. But if $1 \notin C_i$ this means that processor 1 did not send a summary message to i. Thus processor $j = 1$ crashed. □

Lemma 5.5. *In any execution of algorithm Do-UM, if a processor j is in F_i^k of* 425 *some processor i for iteration $k \geq 2$, then either processor j crashes in iteration k or $j \in F_p^{k-1}$ for some $p \in P$.*

Proof. If $j \in F_i^{k-1}$, we are done. Otherwise, by the code, processor i can add processor j to F_i in lines 17, 27, or 28.

Case 1: j is added in line 17 or 27. In this case we have that $j \in m.D$ for 430 some message m sent during iteration k. This implies that $j \in F_p^{k-1}$ of some processor $p \in P$.

Case 2: j is added in line 28. In this case we have that processor j is in $L(\ell)$ but not in C at processor i. If $j \in L(\ell)$ then processor i believes j to be coordinator, and thus processor i sends a report message to j in the Collect 435 stage. If $j \notin C$, this means that processor j does not send a summary message to i. Thus processor j crashed. □

Next we state and prove the following.

Lemma 5.6. *In any execution of algorithm Do-UM, if a processor j is in F_i^k of some processor i for iteration k, then processor crashes in one of the iterations* 440 *$1, 2, ..., k$.*

Proof. By induction on iterations using Lemma 5.4 for the base case, and Lemma 5.5 in the inductive step. □

The above lemma allows us to conclude that the set of active processors never partitions.

445 **Corollary 5.7.** *In any execution of algorithm Do-UM, the set $P - F_i$ for any $i \in P$ includes all non-crashed processors.*

Finally we show that each non-faulty processor makes local progress. This means that the algorithm terminates.

Lemma 5.8. *In any execution of algorithm Do-UM, if processor i does not crash by the end of iteration $k > 0$, then $D_i^{k-1} \subset D_i^k$.*

Proof. This follows from the facts that $D_i^{k-1} \subseteq D_i^k$ (trivially, by the code), and that even if processor i does not receive messages from other processors (e.g., due to coordinator failures) in iteration k, it still performs a task from the set $T - D_i^{k-1}$ and adds it to D. Hence in the worst case $|D_i^k| = |D_i^{k-1}| + 1$. □

Now we give the main result of this section.

Theorem 5.9. *Algorithm Do-UM for n processors and t tasks, using unreliable broadcast, correctly solves the Do-All problem in any execution with up to $n-1$ crashes.*

Proof. Lemma 5.3 shows that a task is never considered done by any processor unless the task was indeed performed by some processor. Lemma 5.8 shows that each processor makes monotone progress. By the failure model, at least one processor does not crash. Thus all non-crashed processors ultimately learn that all tasks are complete, and hence the problem is correctly solved (by $O(n)$ time). □

Remark. Algorithm *Do-UM* solves *Do-All* for any t. This is done "automatically" by having any processor always being able to compute locally a balanced processor-to-task allocation to decide which (single) task it has to perform in a given iteration. There is an alternative approach that can be used when $t > n$. This is done with the help of the conventional "chunking" strategy that partitions the t tasks into n chunks, each containing $\lceil t/n \rceil$ tasks. Now the algorithm is used to perform all n chunks of tasks (instead of individual tasks). The main algorithmic difference for this approach is that each iteration now takes time $\Theta(t/n)$. However the qualitative properties of the algorithm assessed in this section remain invariant.

475 *5.2. Worst-case available processor steps*

We now provide bounds on the performance of the algorithm. We start by proving an upper bound on the available processor steps. Then we show that this upper bound is tight.

5.2.1. Upper bound

Consider an execution α of the algorithm. We know that the algorithm terminates. So let α_z be the last iteration of α. Hence α is made up of z iterations:

$$\alpha = \alpha_1, \alpha_2, \alpha_3, \ldots, \alpha_{z-1}, \alpha_z.$$

480 By Theorem 5.9 it must be the case that there are no tasks left unaccounted after iteration α_z.

Definition 5.10. *In an execution of the algorithm, a live processor considers an iteration* attended *if it receives at least one* `summary` *message.*

Notice that the notion of attended iteration is a local notion.

485 **Definition 5.11.** *In an execution of the algorithm an iteration is called* completely attended *if each live processor receives a* `summary` *message from the same coordinator.*

Definition 5.12. *In an execution of the algorithm an iteration is called* uniquely attended *if there is a unique coordinator.*

490 **Definition 5.13.** *In an execution of the algorithm an iteration is called* good *if*

1. *All live processors send* `report` *messages to a processor c*
2. *And the iteration is completely and uniquely attended by c.*

Notice that in a good iteration the attending coordinator receives a `report`
495 message from every live processor (recall that there are no message failures) and since the phase is completely attended every live processor receives the `summary`

message sent by the attending coordinator. Hence at the end of a good phase the information held by each live processor is consistent. More formally we have the following lemma.

Lemma 5.14. *At the end of a good iteration, for any two live processors i and j we have that $D_i = D_j$ and that $F_i = F_j$.*

Proof. Let k be the good iteration and let c be the attending coordinator. Let i and j be any two live processors, that is any two processor that belong to A^k (the set of live processors at the end of iteration k). Since iteration k is good all live processors send a `report` message to c. Hence the sets D and F computed by c in lines 16 and 17 contain all the information known by any processor in A^k. The sets D and F are sent by c in the `summary` message of the disseminate stage. Since both i and j are alive they will receive the `summary` message sent by c, and hence we have that $D_i = D_j$ after the assignment in line 26 and that $F_i = F_j$ after the assignment in line 27. Moreover since the iteration is completely and uniquely attended, no new processor ids can be included in F at line 28. □

Definition 5.15. *A crash is attributed to the iteration where its effect is perceived, that is, to the iteration where a message that is supposed to be sent by the crashed processor is not received by a live processor.*

Notice that the actual point in time when a crash occurs is not important as long as the crashed processor does not send messages because all changes are local. Hence we could think of crashes as occurring exactly before a processor sends a message (all the actions taken by the processor since the sending of the previous message will be anyway lost).

We use *bad* iteration, to stand for an iteration that is not good. Let S_G be the available processor steps spent during all the good iterations and let S_B be the available processor steps used during all the bad iterations. Then, we have that $S = S_G + S_B$.

525 Since all the good iterations are uniquely and completely attended by a coordinator then all processors have consistent views of the computation (Lemma 5.14). This allows us to evaluate S_G using the same approach used to assess the upper bound on S_a for AN [6].

Lemma 5.16. *The work performed in all the good iterations of an execution of*
530 *Do-UM is* $S_G = O(t + n \log n / \log \log n)$.

Since the proof is basically the same as that of the upper bound on S_a for AN [6] we omit it from the main text. For the benefit of the reader however we provide the proof in the Appendix.

Assessing S_B requires a different approach. This is because of the unreliable
535 broadcast that might cause inconsistencies in the views. The following lemma is immediate by the definition of the available processor steps complexity measure.

Lemma 5.17. *The work in any single bad iteration* α_i *is at most* n_i*, where* n_i *is the number of processors alive at the beginning of iteration* α_i.

The following lemma provides a bound on the total number of iterations in
540 the bad periods.

Lemma 5.18. *In any execution of the algorithm that contains* k *crashes, the total number of bad iterations is at most* $2k$.

Proof. We prove the lemma by induction on the number of crashes in the prefix.

545 Base case: $k = 1$. If the crash happens on the last iteration then the only iteration that is bad is that one. If the crash happens before, say at iteration number x, then iteration x can be bad. The subsequent iteration $x + 1$ might also be bad because two processors might have different local views and thus they might be sending report messages to different coordinators. Since, however,
550 there are no other crashes, iteration $x + 2$ (and any subsequent ones) must be good.

Inductive step: Assume the lemma is true for a fixed value k, we need to prove that is true for $k + 1$. Take any execution and consider the last iteration x where one or more crashes happen. We can apply the inductive hypothesis to
555 the first $x - 1$ iterations of the algorithm. By the inductive hypothesis we know that in the first $x - 1$ iterations there can be at most $2(k - 1)$ bad iterations. The crashes that happen during iteration x might cause iteration x to be bad. Moreover, also iteration $x+1$ can be bad because processors might have different views and thus they might be sending report messages to different coordinators.
560 Since, however, there are no other crashes, iteration $x + 2$ (and any subsequent ones) must be good. □

Corollary 5.19. *In any execution with f failures, the total number of bad iterations is at most $2f$.*

Lemma 5.20. *In any execution with f failures, we have that $S_B = O(f(n-f))$*

Proof. Consider all the bad iterations. By Lemma 5.19 we have that there can be at most $2k$ bad iterations, with $k \leq f$. Moreover we can have exactly $2k$ bad iterations only if each bad iteration with failures is followed by a bad iteration without failures. Roughly speaking this is the worst case scenario where we have one failure rendering two iterations bad. Thus we will consider this worst case scenario to estimate the upper bound: we have $I_1, I_2, \ldots, I_{2k}$, with $k \leq f$, bad iterations where I_{2i-1} is a bad iteration with a failure and I_{2i}, is a bad iteration without failures, for $i = 1, \ldots, k$. Let let n_i be the number of processors alive at the beginning of I_i. Each single iteration I_i might cause at most n_i work. Thus we have that the total work S_B expended in bad iterations is

$$S_B \leq \sum_{i=1}^{2k} n_i.$$

565 Because failures happen in iterations with an odd index, we have that the number of processors decreases at least by one for the subsequent iteration. More specifically, while the first bad iteration I_1 can have the full set of n processors,

the second bad iteration can have at most $n-1$ processors. Similarly, while the third bad iteration I_3 can have at most $n-1$ live processors the fourth bad
570 iteration I_4 can have at most $n-2$ live processors, and so on. Thus the total number of processor steps expended during the bad iterations is

$$\begin{aligned} S_B &\leq \sum_{i=1}^{2k} n_i \\ &\leq n + (n-1) + (n-1) + (n-2) + \ldots + (n-k+1) + (n-k) \\ &= 2kn - k(k-1). \end{aligned}$$

Since k is upper bounded by f we have that $S_B = \Theta(f(n-f))$, as claimed. □

From Lemmas 5.16 and 5.20 we conclude the following.

Theorem 5.21. *In any execution with f failures we have that* $S =$
575 $O\left(t + n\frac{\log n}{\log\log n} + f(n-f)\right)$.

Proof. The total number of steps S spent by the algorithm is given by the sum of steps spent in all iterations that can be partitioned in good and bad. Thus we have that

$$S = S_B + S_G.$$

From Lemma 5.16 we have that

$$S_G = O(t + n\log n / \log\log n),$$

while from Lemma 5.20 we have that

$$S_B = \Theta(f(n-f)).$$

Summing up these two bounds we get the bound on S claimed in the statement of the theorem. □

Remark 1: One might wonder: is the definition of a good iteration too strong? In other words, is it possible to have executions that do not have any good

580 iterations? This is only possible when t is small. According to Lemma 5.19 an adversary can only cause a linear number (in f) of bad iterations and the maximum work that can be done in those iterations is $O(fn)$, thus since t can vary independently of n it might be not possible to perform all tasks during such iterations. Thus for large t, even for worst-case adversary, there must exist 585 executions with some good iterations.

Remark 2: When t is the dominant factor the bound on S given by Theorem 5.21 reduces to $O(t)$. The bound on S provided in [6], namely $O\left(\log f\left(t + p\frac{\log p}{\log\log p}\right)\right)$, for large t, becomes $O(t \log f)$. Since our bound holds also for the setting of [6], when t is the dominant factor, we get a better upper 590 bound on S.

5.2.2. A lower bound for Do-UM

In this section we are going to show an execution with $S = \Omega\left(t + n\frac{\log n}{\log\log n} + f(n-f)\right)$ for $t > n$. Remember that the load balancing rule allocates tasks according to the ids of the outstanding tasks and the ids of the 595 processors giving the ith outstanding task to the ith alive processor.

The run starts with a bad iteration in which processor 1, the sole coordinator, fails in the middle of the send step of the `summary` messages. The `summary` message is received by half of the processors, and therefore the processors get partitioned into two subsets, P_1 and P_2. In particular we set

$$P_1 = \{2, 4, 6, 8, \ldots, n\}$$

$$P_2 = \{3, 5, 7, 9, 11 \ldots, n-1\}.$$

Processors in P_1 receive the `summary` message hence they will have $F = \emptyset$ while processors in P_2 will set $F = \{1\}$. The set of done tasks at this point is for everybody $D = \emptyset$. According to the load balancing rule, processors in P_1 will get a task allocating the tasks over P, which means that processor i, 600 $i = 2, 4, 6, 8, \ldots$, executes task i, while processors in P_2 will get a task allocating the task over $P \setminus \{1\}$, which means that processor i, $i = 3, 5, 7, 9, \ldots$, executes

task $i-1$. This means that task i, $i = 2,4,6,8,\ldots,n$ gets executed by two processors (processor i and processor $i+1$) while tasks $3,5,7,9,\ldots$ don't get executed. This implies that in this iteration we have $n/2$ wasted steps. Phase 2 is attended by processor 2 and 3, as expected by processors in P_2. Notice that also processors in P_1 will receive the summary messages of processors 2 and 3 so for the load balancing rule of the second stage all processors have the same views (although in our definition this second iteration is also bad).

A single failure has caused $n/2$ wasted steps in this first bad period of two iterations. The run now has a good period.

The above pattern (with the appropriate adjustments of the processors and tasks indexes) repeats f times. In all the f times we always have more outstanding tasks than alive processors (recall that $t > n$). For each repetition the total number of alive processors decreases by one, so in the ith repetition there are $n-i$ alive processors and thus $(n-i)/2$ wasted steps. The repetitions of the pattern go on until there are processors than can be crashed.

Having used all the available f failures causing a total of $O(f(n-f))$ wasted steps, the computation now proceeds with a good period until the completion of the tasks. During the good periods the steps spent are $t + n\frac{\log n}{\log\log n}$. Hence for this specific run we have:

$$S = \Theta\left(t + n\frac{\log n}{\log\log n} + f(n-f)\right).$$

5.3. Message analysis

In this section we provide a simple upper bound on the number of messages needed by *Do-UM*. The bound uses a coarse analysis and thus is not tight.

Lemma 5.22. *The number of messages sent in all the good iterations of Do-UM is $M_G \leq 2nt$.*

Proof. By definition in any good phase at most $2n$ messages are sent (at most n `report` messages and at most n `summary` messages). The total number of

good iterations cannot exceed t because in a good iteration at least one task is
625 performed. □

Lemma 5.23. *The number of messages sent in all the bad iterations of Do-UM is* $M_B \leq \frac{fn^2}{2}$.

Proof. By Corollary 5.19 we have that the total number of bad iterations is at most $2f$. In each bad iteration workers might send `report` messages to
630 multiple coordinators which will respond with multiple `summary` message. The total number of messages sent in such a case is $2nc$, where c is the number of coordinators. Due to the layered structure used to elect coordinators, such a number cannot be more than $n/2$. Hence the total number of messages sent in a bad iteration is upper bounded by $n^2/4$. □

635 **Theorem 5.24.** *The number of messages sent in an execution of Do-UM is* $M \leq n(2t + \frac{fn}{2})$.

Proof. Immediate from Lemmas 5.22 and 5.23. □

6. Experimental Evaluation

In the previous section we have analysed the *Do-UM* algorithm providing
640 the worst-case analysis. In this section we are concerned with providing an assessment of the performance in real-life scenarios. In order to evaluate the performance of algorithm *Do-UM*, we have implemented algorithms *Do-UM* and *AN*. Then we have run various experiments under different adversarial assumptions. Before presenting the implementation results, we briefly discuss how
645 this implementation was developed and thereafter where it was executed.

Implementation design. The implementation of both algorithms, *Do-UM* and *AN*, was developed using YALPS [8]. YALPS is an open-source Java library developed by a group of researchers from INRIA institute. The main purpose of YALPS is to facilitate the development, deployment, and testing of distributed

650 applications. More specific, the implementation is written in Java using the YALPS library and the message passing network was an actual physical network formed from a selection of physical nodes which constitute PlanetLab.

PlanetLab [9] is a planetary-scale research network that allows researchers, regardless of their location, to develop, apply, test and evaluate distributed 655 applications. Currently, PlanetLab consists of 1353 nodes at 717 sites and we were able to allocate 150 physical nodes. From those, 128 were used in the executions and the rest were ready to replace any node from the 128 ones that was not acting as expected. In addition, since we were able to allocate the desired number of nodes, we limit the load on any node only to one process. 660 The master computer, that was in charge of (*i*) distributing all the commands to all those physical nodes, (*ii*) collecting back all the data and the logs after an execution, and (*iii*) processing the collected information, was a MacBook Pro OSX with 2.4 GHz Inter Core 2 Duo with 8GB of RAM.

Since algorithm *AN* operates under the assumption of reliable broadcast, 665 we used methods from YALPS that support the use of TCP when exchanging messages in the physical network. In contrast, since algorithm *Do-UM* does not rely on reliable multicast, we used methods from YALPS that support the use of UDP protocol.

In addition, since both algorithms require a *synchronous* setting in order to 670 behave as expected, the big challenge was to synchronize the physical machines that we allocate from PlanetLab and they were forming our physical network. This was achieved by:

1. Selecting carefully nodes from PlanetLab that their processing clocks were mostly identical or in the same range - clock rates range 2.4 - 2.6GHz.
2. 675 Providing a shell script to each node such that after receiving a command from a "master" computer will start the execution on a specific time; this enforced all nodes to start their execution at the same time.
3. Introducing delays in the code in each round in order to allow any "slow" nodes to catch up.

680 4. Collecting and processing the logs after each complete execution to a master computer to validate the fact that all rounds were synchronized.

This implementation is designed to compare the work and the message complexities of the two algorithms. Thus, it is sufficient to use the actual node logs after a successful execution to account for processing steps and exchanged
685 messages.

Failure models for the experiments. We implemented the following failure patterns: *failure free pattern* where failures do not exist, *random failure pattern* where processors fail with some probability, *coordinators failure pattern* where an adversary crashes coordinators, and *lower bound failure pattern* that follows
690 the construction from [10]. The failure free pattern may not be a realistic situation; however its interest is to emphasize the efficiency of the algorithms in the case where processor failures do not exist. The random failure pattern is the failure pattern that is indicative of realistic situations where failures are not correlated. (We remark that correlated failures can actually arise under certain
695 conditions, see for example [29] and thus, as we point out in the directions for future work, this failure scenario is also interesting, although we are not considering it in this paper.) The coordinator failure pattern is of interest because it models a nefarious scenario where crashes target the coordinators. Finally the lower bound failure pattern is of interest because it models a known
700 worst case scenario.

(1) *Failure Free Pattern.* As we previously mentioned, such an execution where processor failures do not exist, nowadays is not a realistic situation. Nonetheless, it is interesting to emphasize the efficiency of both algorithms *Do-UM* and *AN* in the absence of failures. Note that due to the adverse nature of PlanetLab,
705 some failures could occur during the execution; in gathering our results, for this scenario we ignored these cases (we repeated the run).

(2) *Random failures.* Such failures may be expected in realistic executions. Here a processor crashes with probability *error_rate* during the Send step of each stage, and each message multicast by the crashing processor is delivered with

710 probability 0.5, simulating the unreliable broadcast model. The fixed probability of crashes is chosen to be high enough to prevent the algorithm from terminating too quickly, while being low enough to maintain the expectation that all tasks are performed before all processors fail. We term such probability as the *maximum sustainable error rate* and denote it by *mser*. We ran experiments to determine
715 the threshold *mser* such that for $error_rate < mser$ all tasks are *expected* to be completed, while for $error_rate > mser$ all processors are expected to crash before completing all the tasks. For our runs with $t = n$ we approximated *mser* to be about 16%.

(3) *Coordinator failures.* While crashing coordinators is perhaps an unrealis-
720 tic scenario, nevertheless it is useful to understand the impact of coordinator crashes. We chose the failure patterns where the adversary crashes coordinators *before* the Send step of the Disseminate stage. Any active coordinator is crashed until only one layer remains in the layered coordinator structure. This forces the surviving processors at the lowest layer to behave similarly to the all-to-all
725 strategy.

(4) *Coordinator send failures.* This is the same as coordinator failures, except that the adversary crashes coordinators *during* the Send step of the Disseminate stage.

(5) *Lower bound failures.* Here the adversary crashes processors only when
730 they are assigned to tasks in the Compute steps. The construction follows the adversarial strategy $\mathfrak{U}$ of [4] (page 24). The adversary determines the set of undone tasks U, computed as $T - D$ in each iteration (in these scenarios the sets D are the same for all correct processors). Then the adversary chooses $\frac{1}{\log n}|U|$ tasks with the least number of processors assigned to them, and crashes those
735 processors, if any. The adversary continues as long as the number of undone tasks is greater than 1. As soon as only one undone task remains, the adversary allows all remaining processors to perform the task. This adversarial strategy is proved (in [10]) to cause work $\Omega(t + n\frac{\log n}{\log \log n})$. Note that for this adversarial strategy, the processors are crashed only prior to executing the assigned task,
740 and not when sending a message. Thus no message is lost due to the unreliable

broadcast.

Communication reliability. We remark that for algorithm *AN* we used a setting with reliable multicasts. Hence a message sent by processor that fails, either gets completely discarded or it gets delivered to all of its recipients. Instead for algorithm *Do-UM* we used a setting with unreliable multicasts. In this setting a processor can fail during the Send step possibly causing the message to be delivered only to a subset of the recipients. Notice that for the Coordinator failures pattern there is no difference since we assume that the failure happens before the send step so even if broadcast is unreliable the unreliability will never happen (and this is why the results for two algorithms are nearly identical, as will be shown in Figures 4 and 9).

Experimental Evaluation. For our experimental evaluation of algorithms *Do-UM* and *AN* we used $n = t$ parameterization. This is because all *Do-All* algorithms published work complexity become more asymptotically efficient as t grows with respect to n (cf. in algorithm *AN* this is due to the additive term t present in the asymptotic expression). Thus the differences among the algorithms are most evident when $n = t$.

For our experiments we used $n \in \{ 8,16,32,64,128 \}$. The upper limit was chosen for practical reasons based on the physical nodes we were able to allocate and the time it took to run each execution. To make the result measurement more statistically meaningful, for the *Random Failures* patterns and for the *Coordinator Failures* patterns each test was repeated 8 times and the outcome measurements were averaged. For the rest of the scenarios, *Failures Free*; *Coordinator Send Failures*; and *Lower-Bound*; a single run for each n is sufficient because the failures, in case they exist, are deterministic. All the failures were simulated meaning that we did not force actual nodes to crash, but we just considered them as crashed by forcing them not to participate in any future steps of the execution. Due to the adverse conditions in PlanetLab, some nodes could become unresponsive, but with the exception of the Failure free scenario, these nodes did not affect the results.

Finally, as we already mentioned in the implementation design, when running algorithm *AN* we used methods for exchanging messages over the network that support reliable broadcast (TCP), while running algorithm *Do-UM* we used methods that do not ensure reliable broadcast (UDP).

As a next step we present and analyse the results gathered from those executions regarding work and message complexities.

Work complexity. Here we present and discuss work measurements recorded in the executions.

(1) *Failure Free Pattern.* The results are in Figure 2. The analysis is straightforward. In each iteration each processor executes one task, so in each iteration n tasks are executed. In order to execute all tasks $\lceil \frac{t}{n} \rceil$ iterations needed. Hence a total of $n\lceil \frac{t}{n} \rceil$ tasks are executed (in the last iteration some tasks might be executed twice by two different processors). Thus the work complexity is $O(t)$.

For comparison purposes, we note that when broadcast is reliable, the analysis in [6] is readily adapted for use with algorithm *Do-UM* to show that its performance is identical to that of algorithm *AN*. Specifically, the work is $O((t + n \log n/ \log \log n) + \log f)$ where $f < n$ is a bound on the number of processor crashes.

(2) *Random Failures.* The results are in Figure 3. For these simulations we used $error_rate = 12\%$ (this is lower than the approximated $mser = 16\%$). Note that for higher error rates it is possible for all processors to crash, and thus not all tasks may be performed. In such cases the *Do-All* problem is not solved and it does not make sense to evaluate the executions of the algorithm when the problem is not solved. We chose the error rate of 12% because it is somewhat lower than the expected threshold 16%. We expect that tests with different values of the error rate would not substantially change the results (as long as we avoid the possibility of all processors crashing).

Random failures is a relatively benign failure pattern that infrequently impacts all coordinators. The work of algorithm *Do-UM* is substantially the same as the work of algorithm *AN*. In both cases the work is bounded by $2n$ from

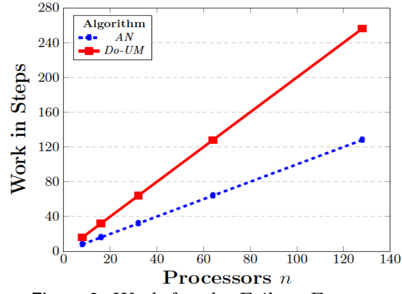

Figure 2: Work for the Failure Free pattern

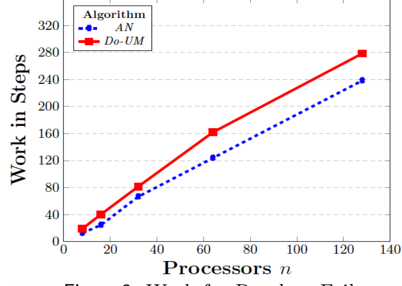

Figure 3: Work for Random Failures

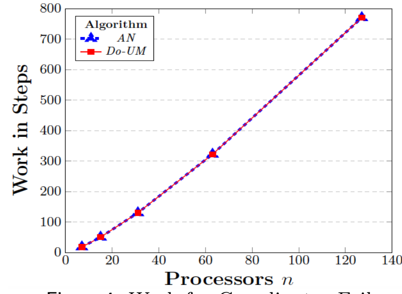

Figure 4: Work for Coordinator Failures

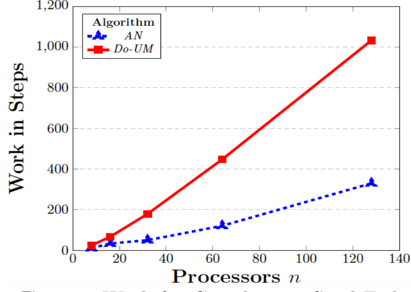

Figure 5: Work for Coordinator Send Failures

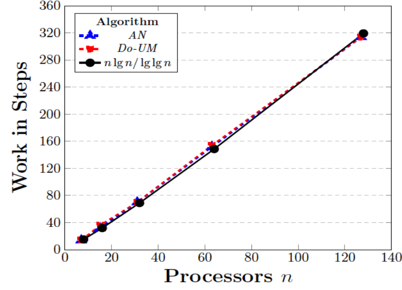

Figure 6: Work for the Lower Bound pattern

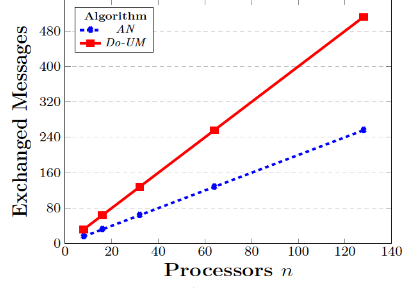

Figure 7: Messages for the Failure Free pattern

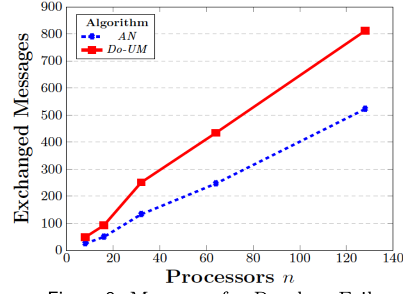

Figure 8: Messages for Random Failures

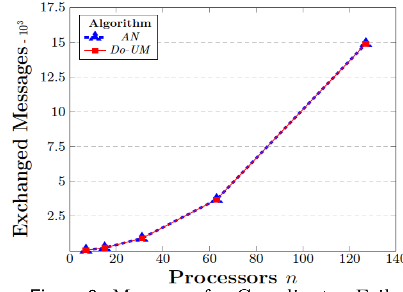

Figure 9: Messages for Coordinator Failures

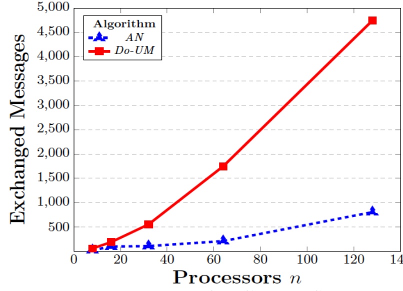

Figure 10: Messages for Coord. Send Failures

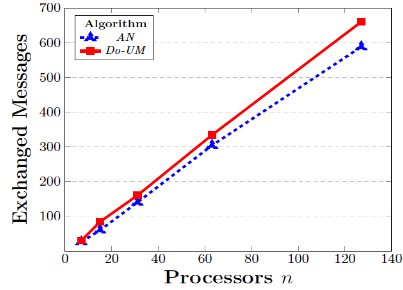

Figure 11: Messages for the Lower Bound pattern

above, although it appears that work grows in a slight superlinear pattern.

(3) *Coordinator Failures.* The results are in Figure 4. As anticipated, the work in this scenario is substantially worse than in the random failure model. This is because coordinator failures are particularly damaging to coordinator-based algorithms. The two algorithms show nearly identical work trend: this is not surprising since the crashes always happen before a Send step and thus there is no difference due to the reliability/unreliability of the multicast. Algorithm *Do-UM* is slightly more efficient than algorithm *AN*, due to its more compact iteration structure.

(4) *Coordinator Send Failures.* The results are in Figure 5 and they illustrate the strength of reliable broadcast. We simulate algorithm *Do-UM* and algorithm *AN*, except that crashes now occur during a coordinator multicast. As expected, the work of both algorithms is better than the scenario where crashes happen before a multicast. However the work of algorithm *AN* is substantially lower. This, because the reliable broadcast used by algorithm *AN* allows the processors to make consistent progress, while algorithm *Do-UM* is disadvantaged by the unreliable broadcast.

(5) *Lower Bound Failures.* The results are in Figure 6. Here we simulate algorithm *Do-UM* and algorithm *AN* each subjected to the lower-bound failure pattern as discussed. We also plot the function $n \log n / \log \log n$ that expresses the lower bound on work [10]. As anticipated, the two coordinator-based algorithms and the all-to-all algorithm track the lower bound closely due to the complete information provided to all workers in each iteration.

Message complexity. We now present and discuss message measurements recorded in executions.

(1) *Failure Free Pattern.* The results are in Figure 7. Similarly to the work complexity analysis, the total number of messages is $2n\lceil \frac{t}{n} \rceil$ since in each iteration $2n$ messages are sent (n *report* messages and n *summary* messages). Thus both the work and message complexities are $O(t)$.

For comparison purposes, we note that when broadcast is reliable, the anal-

ysis in [6] is readily adapted for use with algorithm *Do-UM* to show that its performance is identical to that of algorithm *AN*. Specifically, the message complexity is $O(t + n \log n / \log \log n + fn)$, where $f < n$ is a bound on the number of processor crashes.

835 (2) *Random failures.* The results are in Figure 8. Recall that for these simulations we used *error_rate* = 12%. Although this is a relatively benign failure pattern that infrequently impacts all coordinators, it apparently causes substantial increase in communication in algorithm *Do-UM* as compared to algorithm *AN*. In particular, algorithm *Do-UM* sends up to twice as many messages. Recall 840 that algorithm *AN* has the significant advantage of being able to broadcast reliably, while algorithm *Do-UM* is at a disadvantage, endowed only with unreliable broadcast. We observe that algorithm *Do-UM* is forced to double coordinators earlier than this occurs in algorithm *AN*.

(3) *Coordinator failures.* The results are in Figure 9. The communication 845 burden in this scenario is substantially worse than in the random failure model. Here coordinator failures are particularly damaging, pushing the martingale strategy to its limit, when the workers in the lowest layer of the coordinator structure become coordinators, sending quadratic number of messages. Thus, not surprisingly, the graph has a parabolic nature. As we noted for the work 850 complexity, also for the message complexity the two algorithms behave nearly identical. This, since the crashes always happen before a send step and thus the reliability of the communication does not make any difference.

(4) *Coordinator send failures.* The results are in Figure 10. As anticipated, the communication expense for both algorithms is smaller than above since coordi-855 nators are able to send messages in at least some cases. However, algorithm *AN* is noticeably more efficient due to consistency afforded by the reliable broadcast.

(5) *Lower bound failures.* The results are in Figure 11. Here, the two algorithms are subjected to the lower-bound failure pattern as previously discussed. The messaging trends here are very similar, but with algorithm *Do-UM* incurring 860 somewhat greater expense due to its structure as it executes one more iteration than algorithm *AN*; this is because in algorithm *Do-UM* processors communicate

before performing tasks, while in algorithm *AN* processors communicate after performing tasks, thus more information is communicated in each iteration of algorithm *AN*.

Final remarks. The experimental evaluation assesses the value of the proposed algorithm *Do-UM* showing that in real life scenarios it achieves performances comparable with those of *AN*. The drawbacks that emerge from the theoretical analysis and from the experimental data are due to the intrinsic disadvantage of the scenario in which *Do-UM* is supposed to run, that is, in systems with unreliable message passing multicast mechanisms. To cope with this unfavorable scenario the algorithm spends many more messages and processors steps due to the doubling of the number of coordinators. This clearly induces an overhead which, however, as the experimental data shows, is acceptable since the overall performance, both with respect to work and with respect to messages is comparable to that of the algorithm that runs in the more favorable setting with reliable multicast.

7. Conclusions

We presented a new algorithm for cooperative computing in message-passing settings with crash-prone processors. The algorithm allows participating processors to collaboratively execute a set of identical and idempotent tasks despite possible processor failures. Previous solutions to this problem assume multicast reliability: if a processor crashes while multicasting a message, then it is guaranteed that either the message is delivered to nobody or to all of its intended recipients. Without this assumption the message could be delivered only to some of the intended recipients causing processors to have different information about the system. The algorithm that we propose is able to cope with unreliable multicast. We have analysed the algorithm both from a theoretical point of view, proving its correctness and its theoretical efficiency, and from an experimental point view, implementing it and running several tests to assess its performance. The results of the tests show that the algorithm behaves well:

even if multicast is unreliable, its performance is comparable to those algorithms running under the assumption of reliable multicast.

Interesting directions for future work include a deeper experimental investigation of the practical behaviour of the algorithm under a variety of settings that one can obtain by varying the parameters of the system, considering specific job types (e.g., using real traces and workloads) or considering other failure scenarios (e.g., correlated failures [29]).

Acknowledgments.. This work is supported in part by the NSF award 1017232, by research grants of the University of Cyprus (ED2016-CG) and by the Italian MIUR PRIN projects fund.

References

[1] S. Davtyan, R. D. Prisco, C. Georgiou, A. A. Shvartsman, Coordinated cooperative work using undependable processors with unreliable broadcast, in: Proceedings of the 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP 2014), 2014, pp. 17–26.

[2] Internet primenet server.
URL `http://mersenne.org/ips/stats.html`

[3] SETI@home.
URL `http://setiathome.ssl.berkeley.edu/`

[4] C. Georgiou, A. A. Shvartsman, Do-All Computing in Distributed Systems: Cooperation in the Presence of Adversity, Springer-Verlag, 2008.

[5] R. De Prisco, A. Mayer, M. Yung, Time-optimal message-efficient work performance in the presence of faults, in: Proceedings of the 13th ACM Symposium on Principles of Distributed Computing (PODC 1994), 1994, pp. 161–172.

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

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

[8] Yalps.
URL http://yalps.gforge.inria.fr

[9] Planetlab.
URL https://www.planet-lab.org/

[10] P. Kanellakis, A. Shvartsman, Fault-Tolerant Parallel Computation, Kluwer Academic Publishers, 1997.

[11] Z. Kedem, K. Palem, A. Raghunathan, P. Spirakis, Combining tentative and definite executions for dependable parallel computing, in: Proceedings of the 23rd ACM Symposium on Theory of Computing (STOC 1991), 1991, pp. 381–390.

[12] J. Groote, W. Hesselink, S. Mauw, R. Vermeulen, An algorithm for the asynchronous Write-All problem based on process collision, Distributed Computing 14 (2) (2001) 75–81.

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

[14] D. Alistarh, M. A. Bender, S. Gilbert, R. Guerraoui, How to allocate tasks asynchronously, in: Proc. of the 53rd IEEE Symp. on Foundations of Computer Science (FOCS 2012), 2012, pp. 331–340.

[15] D. Alistarh, J. Aspnes, M. A. Bender, R. Gelashvili, S. Gilbert, Dynamic task allocation in asynchronous shared memory, in: Proc. of the 25th ACM-SIAM Symposium on Discrete Algorithms (SODA 2014), 2014, pp. 416–435.

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

[17] B. Chlebus, D. Kowalski, A. Lingas, The Do-All problem in broadcast networks, Distributed Computing 18 (6) (2006) 435–451.

[18] Z. Galil, A. Mayer, M. Yung, Resolving message complexity of byzantine agreement and beyond, in: Proceedings of the 36th IEEE Symposium on Foundations of Computer Science (FOCS 1995), 1995, pp. 724–733.

[19] B. Chlebus, D. R. Kowalski, Randomization helps to perform independent tasks reliably, Random Struct. Algorithms 24 (1) (2004) 11–41.

[20] K. M. Konwar, S. Rajasekaran, A. A. Shvartsman, Robust network supercomputing with unreliable workers, J. Parallel Distrib. Comput. 75 (2015) 81–92.

[21] S. Davtyan, K. M. Konwar, A. Russell, A. A. Shvartsman, Dealing with undependable workers in decentralized network supercomputing, Theor. Comput. Sci. 561 (2015) 96–112.

[22] C. Georgiou, D. R. Kowalski, On the competitiveness of scheduling dynamically injected tasks on processes prone to crashes and restarts, J. Parallel Distrib. Comput. 84 (2015) 94–107.

[23] S. Dolev, R. Segala, A. Shvartsman, Dynamic load balancing with group communication, Theoretical Computer Science 369 (1–3) (2006) 348–360.

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

[25] D. Kowalski, A. Shvartsman, Performing work with asynchronous processors: message-delay-sensitive bounds, Information and Computation 203 (2) (2005) 181–210.

970 [26] C. Georgiou, D. Kowalski, A. Shvartsman, Efficient gossip and robust distributed computation, Theoretical Computer Science 347 (1) (2005) 130–166.

[27] D. Kowalski, P. Musial, A. Shvartsman, Explicit combinatorial structures for cooperative distributed algorithms, in: Proceedings of the 25th Interna-
975 tional Conference on Distributed Computing Systems (ICDCS 2005), 2005, pp. 48–58.

[28] A. Fernandez, C. Georgiou, A. Russell, A. Shvartsman, The Do-All problem with Byzantine processor failures, Theoretical Computer Science 333 (3) (2005) 433–454.

980 [29] J. Lu, H. Shen, A low-cost multi-failure resilient replication scheme for high data availability in cloud storage, in: Proceedings of the 23rd IEEE International Conference on High Performance Computing, Data, and Analytics (HiPC 2016), 2016.

Appendix

985

Appendix A. Proof of Lemma 5.16

Proof. Consider all the iterations in good periods and call them $\alpha_{i_1}, \alpha_{i_2}, ..., \alpha_{i'_z}$, with $i'_z \leq\!< z$. Let u_{α_i} be the number of undone tasks at the beginning of iteration α_i, and n_{α_i} be the number of alive processors at the beginning of iteration α_i, for each $i = i_1, i_2, \ldots, i'_z$. Partition the iterations in two groups:

990 GROUP 1: All attended iterations α_i such that $n_{\alpha_i} \leq u_{\alpha_i}$. The load balancing rule assures that at most one processor is assigned to a task. Hence the available processor steps used in this case can be charged to the number of tasks executed which is at most $t + f \leq t + n$. Hence $S_1 = O(t + n)$.

GROUP 2: All attended iterations in which $n_{\alpha_i} > u_{\alpha_i}$. We let $d(n)$ stand for

995 $\log n / \log\log n$. We consider the following two subcases.

SUBCASE 2.1: All attended iterations α_i after which $u_{\alpha_{i+1}} < u_{\alpha_i}/d(n)$. Since $u_{\alpha_{i+1}} < u_{\alpha_i} < n_{\alpha_i} < n$ and iteration α_w is the last iteration for which $u_{\alpha_w} > 0$, it follows that Subcase 2.1 occurs $O(\log_{d(n)} n)$ times. The quantity $O(\log_{d(n)} n)$ is $O(d(n))$ because $d(n)^{d(n)} = \Theta(n)$. No more than n processors complete such iterations, therefore the part $S_{2.1}$ of S_G spent in this case is

$$S_{2.1} = O\left(n \frac{\log n}{\log\log n}\right).$$

SUBCASE 2.2: All attended iterations α_i after which $u_{\alpha_{i+1}} \geq u_{\alpha_i}/d(n)$. Consider a particular iteration α_i. Since in this case $p_{\alpha_i} > u_{\alpha_i}$, by the load balancing rule at least $\lfloor \frac{n_{\alpha_i}}{u_{\alpha_i}} \rfloor$ but no more than $\lceil \frac{n_{\alpha_i}}{u_{\alpha_i}} \rceil$ processors are assigned to each of the u_{α_i} unaccounted tasks. Since $u_{\alpha_{i+1}}$ tasks remain unaccounted after iteration α_i,

1000 the number of processors that failed during this iteration is at least

$$u_{\alpha_{i+1}} \left\lfloor \frac{n_{\alpha_i}}{n_{\alpha_i}} \right\rfloor \quad \geq \quad \frac{u_{\alpha_i}}{d(n)} \cdot \frac{n_{\alpha_i}}{2u_{\alpha_i}} = \frac{n_{\alpha_i}}{2d(n)}.$$

Hence, the number of processors that proceed to iteration α_{i+1} is no more than

$$n_{\alpha_i} - \frac{n_{\alpha_i}}{2d(n)} = n_{\alpha_i}(1 - \frac{1}{2d(n)}).$$

Let $\alpha_{i_0}, \alpha_{i_1}, ..., \alpha_{i_k}$ be the attended iterations in this subcase. Since the number of processor in iteration α_{i_0} is at most n, the number of processors alive in iteration α_{i_j} for $j > 0$ is at most $n(1 - \frac{1}{2d(n)})^j$. Therefore the part $S_{2.2}$ of S_G
1005 spent in this case is bounded as follows:

$$S_{2.2} \leq \sum_{j=0}^{k} n\left(1 - \frac{1}{2d(n)}\right)^j \leq \frac{n}{1-(1-\frac{1}{2d(n)})} = n \cdot 2d(n) = O(n \cdot d(n)).$$

Summing up the contributions of all the cases considered we get

$$S_G = S_1 + S_{2.1} + S_{2.2} = O\left(t + n\frac{\log n}{\log \log n}\right),$$

as claimed. □