Coordinated Cooperative Work Using Undependable Processors with Unreliable Broadcast

Seda Davtyan
University of Connecticut
371 Fairfield Way, U-4155
Storrs, CT 06269, USA
seda@engr.uconn.edu

Roberto De Prisco
Università di Salerno
84084 Fisciano (SA)
Italy
robdep@unisa.it

Chryssis Georgiou
University of Cyprus
1678 Nicosia
Cyprus
chryssis@ucy.ac.cy

Alexander A. Shvartsman
University of Connecticut
371 Fairfield Way, U-4155
Storrs, CT 06269, USA
aas@cse.uconn.edu

***Abstract*—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 a distributed environment—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. This paper presents a new message-passing algorithm for distributed cooperative work in synchronous settings where processors may crash, and where any broadcasts performed by crashing processors are unreliable. We specify the algorithm, prove that it is correct, and perform extensive simulations that show that its performance is close to similar algorithms that use reliable broadcast, and that its work compares favorably to the relevant lower bounds.**

***Keywords*-distributed algorithms; fault-tolerance; processor crashes; unreliable broadcast; task computing.**

I. Introduction

Cooperative 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. [1], [2]. 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 the 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*, where the goal is to use n processors to perform t tasks in the presence of adversity [3].

To remove the troublesome assumption of having an infallible master, coordinator-based solutions have been proposed for the *Do-All* problem. Here the approach is to appoint coordinators 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 [4].

Thus, it is advantageous to explore solutions that allow multiple concurrent coordinators. Using multiple coordinators raises a 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, 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 [5]. 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, the algorithm in [5] 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.

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 the unreliable broadcast. The important attributes of target algorithms must also be conceptual simplicity and ease of

implementation.

Prior and Related Work. The *Do-All* problem has been studied in a variety of settings, e.g., in *shared-memory* models [6]–[11], in *message-passing* models [3]–[5], [12]–[15] and in *partitionable networks* [16], [17]. *Do-All* was initially formulated in the shared-memory model with the efficiency of algorithms assessed using the *available processor steps* work complexity measure S [6] that we consider in this work. The lower bound on work for *Do-All* was shown to be $S = \Omega(t + n \log n / \log \log n)$ for synchronous crash-prone processors, and this bound holds both for the shared-memory model and for message-passing model [6]. A comprehensive treatment of the *Do-All* problem in the message-passing setting is given in [3]; basic techniques used in solving *Do-All* can be found in [18].

Dwork, Halpern, and Waarts [12] introduced the *Do-All* problem in the message-passing model. They developed several deterministic algorithms for synchronous crash-prone processors. They evaluated the efficiency of their solutions using the *task-oriented* complexity measure that accounts only for task executions. Most work in the message-passing model focuses on synchronous models (for an example of an asynchronous setting see [19]). The work [4] gave an algorithm for *Do-All* for the synchronous setting with processor crashes using the available processor steps work complexity. They present 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 ($n \ll t$). The authors also show a work lower bound $\Omega(t + (f + 1)n)$ for any algorithm using the stage-checkpoint strategy, this bound being 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 [20] developed a *Do-All* algorithm that beats the lower bound shown in [4]; 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 [21] show how to construct the needed permutations efficiently, yet the algorithm in [20] remains complex and hard to implement.

The paper [5] presents algorithm *AN* whose work also beats the lower bound of [4] 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 phases. Initially there is a single coordinator. If it fails in some phase, then two processors become coordinators in the next phase. If they both fail, then four processors act as coordinators, and so on. If in a given phase at least one of the coordinators survives the phase, then the next phase 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 [18]).

This problem has also been considered in other models of failure, in particular in models with Byzantine failures [22]. 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 round to verify that a certain number of tasks were performed correctly.

Contributions. 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 [5]). Such *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* [5]. We present an iterative algorithm that is simpler than algorithm *AN* in that each iteration consists of two send-receive-compute rounds as compared to three similar rounds 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 [5]).

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* [18]).

3. We perform an extensive simulation study that examines the performance of algorithm *Do-UM* (with unreliable broadcast) in various adversarial settings, and we compare its performance to that of algorithm *AN* (with reliable broadcast) and that of all-to-all algorithm (with unreliable broadcast). The simulation study shows that algorithm *Do-UM* is reasonably efficient in these adversarial settings. In particular, we show that when subjected to a particular strong adversary its performance is close to what is anticipated by the corresponding lower bound.

We emphasize that in our simulation studies we *do not* expect algortihm *Do-UM* to outperform algorithm *AN*. This is because algorithm *AN* uses a *very strong* assumption that *reliable broadcast* is available. In all simulations algorithm *AN* relies on this assumption, while the new algorithm *does not* rely on this, as it uses unreliable broadcast. However, algorithm *AN* presents an idealistic benchmark against which we compare algorithm *Do-UM*. We do note that both algorithms have the same complexity when the broadcast is reliable.

Document structure. In Section II we give the model of computation, definitions, and measures of efficiency. Section III presents our algorithm. In Section IV we show the correctness of the algorithm. In Section V we discuss algorithm performance and the simulation results. We conclude in Section VI.

II. 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 [3].

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 *round* 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 multcasts, 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.

Measures of efficiency. 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. 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.

III. Algorithm Description

We now present algorithm *Do-UM*. Here 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 rounds, a Collect round and a Disseminate round. In the Collect round each processor sends a `report` message to every coordinator. Every processor that receives a `report` message in this round updates its knowledge according to the information contained in the messages. In the subsequent Disseminate round, 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 round 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* [5]. 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 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 round of the iteration. Note that the case of "unexpected" `summary` messages never arises because every processor 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 the tasks that the processor knows to be complete, and F, the set of pids of the processors that the processor knows as crashed. 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 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 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 $k < 2^h - 1$.

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 $L = \langle q_1, q_2, \ldots, q_{13} \rangle$ looks like

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, each iteration consisting of two rounds. Recall that a round consists of a Send step, a Receive step, and a Compute step. The pseudocode for the algorithm is in Figure 1. Both 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) tasks 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 round. In the Send step each processor sends a `report` message to the coordinators. In the Receive step processors receive the `report` messages sent in the Send step. (The messages sent in the very first round 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, using the information received in the messages each processor updates its D and F.

Disseminate round. In the Send step, any processor that either considers itself a coordinator, or receives a `report` message in the Collect round, sends `summary` messages to all non-crashed processors. In the Receive step processors receive the `summary` messages sent in the Send step.

In the Compute step, 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.

IV. Correctness of Algorithm *Do-UM*

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 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 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, 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.

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).

Algorithm *Do-UM* for processor p_i

```
 1:  external T, P
 2:  D : 2^T init ∅ /* set of done tasks */
 3:  F : 2^P init ∅ /* set of crashed processors */
 4:  L : 2^P init P /* coordinator layered structure */
 5:  ℓ : ℕ^≥0 init 0 /* current level in L */
 6:  C : 2^P init ∅ /* last active coordinators */
 7:  R : 2^M init ∅ /* set of received report messages */
 8:  S : 2^M init ∅ /* set of received summary messages */

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

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

Lemma 4.1: 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 round. 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 4.2: 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 4.3: 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 iterations using Lemma 4.1 for the base case and Lemma 4.2 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 4.4: 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 round 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 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 4.5: In any execution of algorithm *Do-UM*, if a processor j is in F_i^k of 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 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 round. 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 4.6: 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 $1, 2, ..., k$.

Proof: By induction on iterations using Lemma 4.4 for the base case, and Lemma 4.5 in the inductive step. ■

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

Corollary 4.7: 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 4.8: 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 worse case $|D_i^k| = |D_i^{k-1}| + 1$. ■

Now we give the main result of this section.

Theorem 4.9: 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 4.3 shows that a task is never considered done by any processor unless the task was indeed performed by some processor. Lemma 4.8 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 thus the time complexity of the algorithm is $O(n)$ and the algorithm terminates. ■

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. There is an alternative approach for dealing with $t > n$. 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.

V. Simulations and Performance Analysis

In order to evaluate the performance of algorithm *Do-UM*, we developed a simulator for algorithm *Do-UM*, algorithm *AN*, and the all-to-all broadcast algorithm. Before presenting the simulation results, we discuss the analysis of algorithm *Do-UM* for the failure-free case and for reliable broadcast.

Failure-free case. In the failure free case the analysis is easy. In each iteration each processor executes one task, so in each iteration n tasks are executed. In order to execute all tasks $\lceil t/n \rceil$ iterations are needed. Hence a total of $n\lceil t/n \rceil$ tasks are executed (in the last iteration some tasks might be executed twice by two different processors). Similarly, the total number of messages is $2n\lceil t/n \rceil$ since in each iteration $2n$ messages are sent (n `report` messages and n `summary` messages). Thus both the work and message complexities are $O(t)$.

Using reliable broadcast. For comparison purposes, we note that when broadcast is reliable, the analysis in [5] 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 $O((t+n\log n/\log\log n)\log f)$ and 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.

Next we present our approach to simulation and the simulation results.

Simulator design. We simulated the algorithms (algorithm *Do-UM*, algorithm *AN*, and the all-to-all broadcast algorithm) in a variety of adversarial settings for the purpose of comparing their performance. The simulator is written in C++ and the message passing network is simulated using a shared array of messages. The dummy simulated tasks consist of storing values in shared memory. The simulation is designed to compare work and messaging expense of the three algorithms, thus it is sufficient to use discrete counters to account for processing steps and messages. Were the algorithms actually implemented, the main distinction would be in the work performance, where instead of counting abstract unit steps, the cost of actual steps will include the time needed to perform real tasks, to do local bookkeeping, and deliver and process messages.

All simulations were run on a multicore Linux server with $X5650$ Intel Xeon and a Mac OSX with 2.3 GHz Intel Core i7.

Failure models for simulations. We implemented the following failure patterns: *random failure pattern* where processors fail with some probability, *coordinators failure pattern* where an adversary crashes coordinators, and *lower bound failure pattern* that follows the construction from [6]. The random failure pattern is the failure pattern that is indicative of realistic situations where failures are not correlated. 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 worst case scenario.

(1) *Random failures.* Such failures may be expected in realistic executions. Here a processor crashes with probability *error_rate* during the Send step of each round, and each message multicast by the crashing processor is delivered with 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 the *maximum sustainable error rate* and denote it by *mser*. We ran experiments to determine 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 simulations with $t=n$ we approximated *mser* to be about 16%.

(2) *Coordinator failures.* While crashing coordinators is perhaps an unrealistic 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 round. 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 strategy.

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

(4) *Lower bound failures.* Here the adversary crashes processors only when they are assigned to tasks in the Compute steps. The construction follows the adversarial strategy $\mathfrak{U}$ of [3] (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 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 [6]) 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, and not when sending a message. Thus no message is lost due to the unreliable broadcast.

Simulations. For our simulations of algorithm *Do-UM*, algorithm *AN*, and all-to-all broadcast algorithm we used $n=t$ parameterization. This is because all *Do-All* algorithms with 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 simulation runs we used $n\in\{128,256,512,1024,2048,4096,8192\}$. The upper limit was chosen for practical reasons based on the time it took to run each simulation. To make the result measurement more statistically meaningful, for the random failures pattern and for the coordinator failures patterns each test was repeated 100 times and the outcome measurements were averaged. For the lower-bound pattern a single run for each n is sufficient because the failures are deterministic.

When simulating algorithm *AN* we always used reliable broadcast that it was designed for (recall that this is a strong assumption). Next we present the results referring to Figures 2 to 9 (the left column of figures reports the work observations and the right column reports the corresponding messaging observations).

Work complexity. Here we present and discuss work mea-

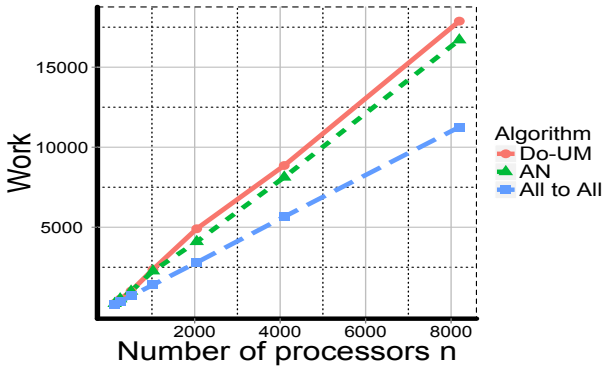

Figure 2. Work for the random failure pattern

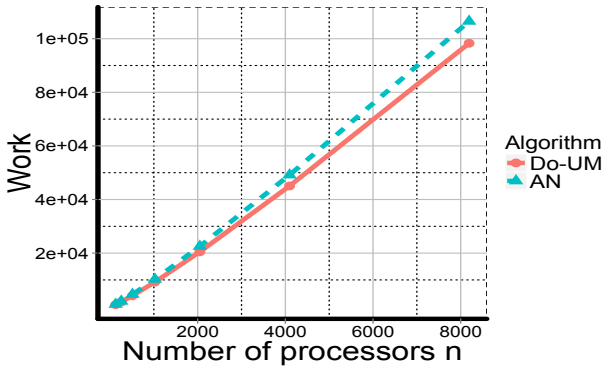

Figure 3. Work for the coordinator failure pattern

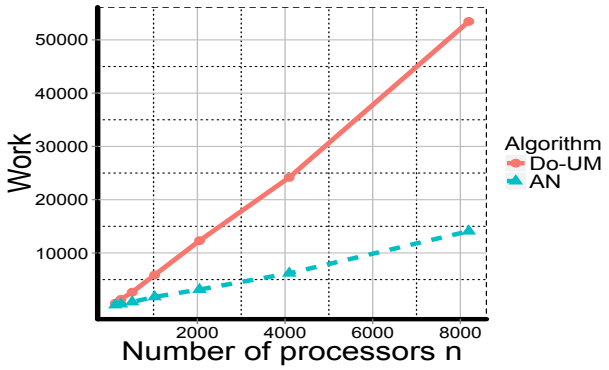

Figure 4. Work for the coordinator send failure pattern

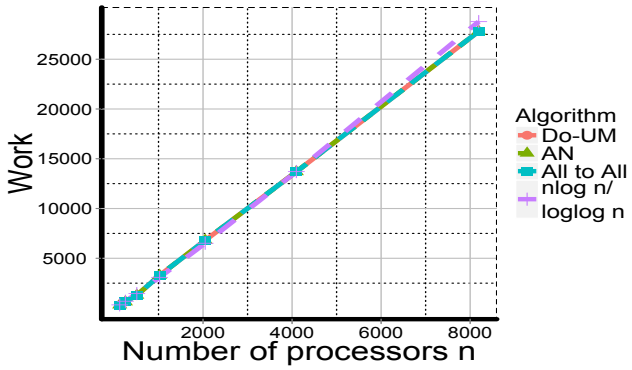

Figure 5. Work for the lower bound pattern

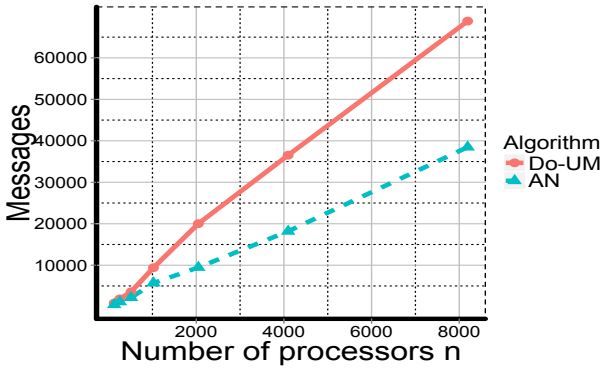

Figure 6. Messages for the random failure pattern

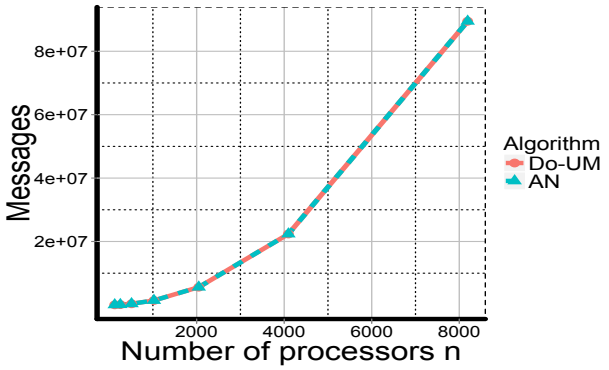

Figure 7. Messages for the coordinator failure pattern

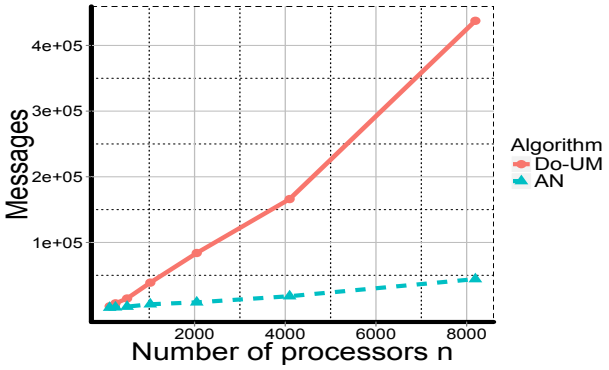

Figure 8. Messages for the coordinator send failure pattern

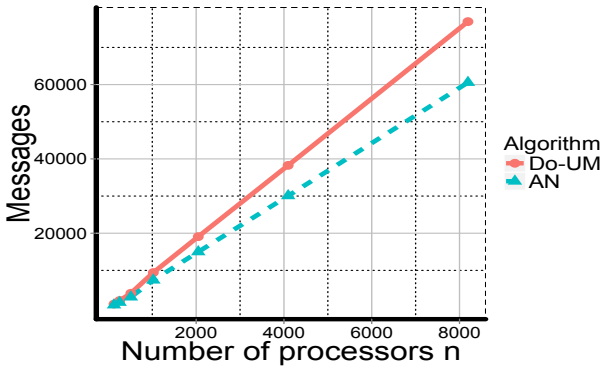

Figure 9. Messages for the lower bound pattern

surements recorded in simulations.

(1) *Random failures.* The results are in Figure 2. 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 that all processors crashing).

All three algorithms were simulated. As anticipated, the 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*. The work of all-to-all algorithm is slightly lower; this is also expected because all information is shared by all processors in each iteration. In all cases the work is bounded by $2n$ from above, although it appears that work grows in a slight superlinear pattern.

(2) *Coordinator failures.* The results are in Figure 3. Here we simulate only the coordinator-based algorithm *Do-UM* and algorithm *AN* (all-to-all algorithm has no coordinators). 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; algorithm *Do-UM* is slightly more efficient, by about the work spent in one iteration, due to its more compact iteration structure.

(3) *Coordinator send failures.* The results are in Figure 4 and they illustrate the strength of reliable broadcast. As above, we again simulate algorithm *Do-UM* and algorithm *AN*, except that crashes occur during coordinator multicasts. As anticipated, the work of both algorithms is better than for the scenario with crashes before multicasts, however the work of algorithm *AN* is substantially lower. It can be observed that this is 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.

(3) *Lower bound failures.* The results are in Figure 5. Here we simulate algorithm *Do-UM*, algorithm *AN*, and all-to-all algorithm, 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 [6]. 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 simulations. We present only the results for algorithm *Do-UM* and algorithm *AN* because the message complexity of all-to-all broadcast is unacceptably high—it is literally off-the-chart (the main purpose of that algorithm is to use it in work comparisons).

(1) *Random failures.* The results are in Figure 6. 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 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*.

(2) *Coordinator failures.* The results are in Figure 7. As anticipated, the communication 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 and both algorithms have nearly identical messaging pattern.

(3) *Coordinator send failures.* The results are in Figure 8. As anticipated, the communication expense for both algorithms is smaller than above because coordinators are able to send in at least some cases. However algorithm *AN* is noticeably more efficient due to consistency afforded by the reliable broadcast.

(4) *Lower bound failures.* The results are in Figure 9. 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 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*.

VI. Conclusions

We presented a new algorithm for cooperative computing in message-passing settings with crash-prone processors. Here the participating processors must perform a collection of tasks despite failures. The algorithm pursues a coordinator-based approach, where multiple concurrent coordinators direct the work of all processors in a way that helps balance the load on all processors. The algorithm *does not assume reliable broadcast* as some prior coordinator-based algorithms—this means that if a processor crashes during a broadcast, then only some arbitrary subset of processors may receive the message. We prove that the algorithm solves the distributed cooperation problem, and that

the entire set of non-crashed processors is able to cooperate during the computation. We then simulate our algorithm in different failure settings, and compare its performance with an algorithm that depends on reliable broadcast, and a benchmark algorithm that uses all-to-all broadcast. The simulations show that although our algorithm does not rely on reliable broadcast, its performance is comparable to that of the algorithm that requires reliable broadcast. In comparison with the all-to-all broadcast algorithm, our algorithm has slightly worse work complexity but substantially better message complexity.

Future work will focus on developing more effective algorithms, and on analytical assessments of performance.

Acknowledgments.: This work is supported in part by the NSF award 1017232, by the Cyprus Research Promotion Foundation grant ΤΠΕ/ΠΛΗΡΟ/0609(BE)/05 and by the Italian MIUR PRIN projects fund.

REFERENCES

[1] "Internet primenet server," http://mersenne.org/ips/stats.html.

[2] "SETI@home," http://setiathome.ssl.berkeley.edu/.

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

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

[5] B. Chlebus, R. De Prisco, and A. Shvartsman, "Performing tasks on restartable message-passing processors," *Distributed Computing*, vol. 14, no. 1, pp. 49–64, 2001.

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

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

[8] J. Groote, W. Hesselink, S. Mauw, and R. Vermeulen, "An algorithm for the asynchronous Write-All problem based on process collision," *Distributed Computing*, vol. 14, no. 2, pp. 75–81, 2001.

[9] R. Anderson and H. Woll, "Algorithms for the certified Write-All problem," *SIAM Journal of Computing*, vol. 26, no. 5, pp. 1277–1283, 1997.

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

[11] D. Alistarh, J. Aspnes, M. A. Bender, R. Gelashvili, and S. Gilbert, "Dynamic task allocation," in *Proc. of the* 25^{th} *ACM-SIAM Symposium on Discrete Algorithms (SODA 2014)*, 2014.

[12] C. Dwork, J. Halpern, and O. Waarts, "Performing work efficiently in the presence of faults," *SIAM Journal on Computing*, vol. 27, no. 5, pp. 1457–1491, 1998.

[13] B. Chlebus, D. Kowalski, and A. Lingas, "The Do-All problem in broadcast networks," *Distributed Computing*, vol. 18, no. 6, pp. 435–451, 2006.

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

[15] B. Chlebus and D. R. Kowalski, "Randomization helps to perform independent tasks reliably," *Random Struct. Algorithms*, vol. 24, no. 1, pp. 11–41, 2004.

[16] S. Dolev, R. Segala, and A. Shvartsman, "Dynamic load balancing with group communication," *Theoretical Computer Science*, vol. 369, no. 1–3, pp. 348–360, 2006.

[17] C. Georgiou, A. Russell, and A. Shvartsman, "Work-competitive scheduling for cooperative computing with dynamic groups," *SIAM Journal on Computing*, vol. 34, no. 4, pp. 848–862, 2005.

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

[19] D. Kowalski and A. Shvartsman, "Performing work with asynchronous processors: message-delay-sensitive bounds," *Information and Computation*, vol. 203, no. 2, pp. 181–210, 2005.

[20] C. Georgiou, D. Kowalski, and A. Shvartsman, "Efficient gossip and robust distributed computation," *Theoretical Computer Science*, vol. 347, no. 1, pp. 130–166, 2005.

[21] D. Kowalski, P. Musial, and A. Shvartsman, "Explicit combinatorial structures for cooperative distributed algorithms," in *Proceedings of the* 25^{th} *International Conference on Distributed Computing Systems (ICDCS 2005)*, 2005, pp. 48–58.

[22] A. Fernandez, C. Georgiou, A. Russell, and A. Shvartsman, "The do-all problem with byzantine processor failures," *Theoretical Computer Science*, vol. 333, no. 3, pp. 433–454, 2005.