Available online at www.sciencedirect.com
SCIENCE DIRECT®
Theoretical Computer Science 333 (2005) 433–454
Theoretical Computer Science
www.elsevier.com/locate/tcs

The Do-All problem with Byzantine processor failures

Antonio Fernández[a,1], Chryssis Georgiou[b,2], Alexander Russell[c,*,3], Alex A. Shvartsman[c,d,4]

[a] *GSyC, Universidad Rey Juan Carlos, 28933 Móstoles, Spain*
[b] *Department of Computer Science, University of Cyprus, 75 Kallipoleos Str., P.O. Box 20537, CY-1678, Nicosia, Cyprus*
[c] *Department of Computer Science and Engineering, University of Connecticut, 371 Fairfield Rd., Unit 2155, Storrs, CT 06269, USA*
[d] *Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, MA 02139, USA*

Received 6 October 2003; received in revised form 30 April 2004; accepted 26 June 2004

Abstract

Do-All is the abstract problem of using *n* processors to cooperatively perform *m* independent tasks in the presence of failures. This problem and its derivatives have been a centerpiece in the study of trade-offs between efficiency and fault-tolerance in cooperative computing environments. Many algorithms have been developed for Do-All in various models of computation, including message-passing, partitionable networks, and shared-memory models under a variety of failure models.

This work initiates the study of the *Do-All* problem for synchronous message-passing processors prone to *Byzantine* failures. In particular, upper and lower bounds are given on the complexity of

*Corresponding author.
E-mail addresses: anto@gsyc.escet.urjc.es (A. Fernández), chryssis@ucy.ac.cy (C. Georgiou), acr@cse.uconn.edu (A. Russell), aas@cse.uconn.edu (A.A. Shvartsman).

[1] Partially supported by the Spanish MCyT under grant TIC2001-1586-C03-01, the Comunidad de Madrid under grant 07T/0022/2003, and the Universidad Rey Juan Carlos under grant PPR-2003-37. Work done while at the University of Connecticut.
[2] Work done in part while at the University of Connecticut.
[3] The work of this author is supported in part by the NSF CAREER Award 0093065 and NSF grants 0220264, 0218443, 0121277, and 0311368.
[4] Partially supported by the NSF CAREER Award 9984778 and the NSF Grants 9988304, 0121277, and 0311368.

doi:10.1016/j.tcs.2004.06.034

Do-All for several cases: (a) the case where the maximum number of faulty processors f is known a priori, (b) the case where f is not known, (c) the case where a task execution can be verified (without re-executing the task), and (d) the case where task executions cannot be verified. The efficiency of algorithms is evaluated in terms of work and message complexities. The work complexity accounts for all computational steps taken by the processors and the message complexity accounts for all messages sent by the processors during the computation. The work and messages of a faulty processor are counted only until the processor fails to follow the algorithm. It is shown that in some cases obtaining work $\Theta(mn)$ is the best one can do. It is also shown that in certain cases communication cannot help improve work efficiency.

Keywords: Distributed cooperation; Independent tasks; Byzantine failures; Work complexity

1. Introduction

The ability to effectively cooperate on common tasks in a decentralized setting is key to solving many computation problems ranging from distributed search (e.g., SETI [20]) to distributed simulation (e.g., [7]) and multi-agent collaboration (e.g., [1,26]). Do-All, an abstraction of such cooperative activity, is the problem of using n processors to cooperatively perform m independent tasks in the presence of failures. The Do-All problem can be used to study trade-offs between efficiency and fault-tolerance in cooperative computing, and is considered to be fundamental in the research on the complexity of fault-tolerant distributed computation [10,17]. Variations of this problem have been studied in shared-memory models (*Write-All*) [18,19,24], in message-passing models [8,10,12], and in partitionable networks (*Omni-Do*) [9,15,22]. Solutions for Do-All must perform all tasks efficiently in the presence of specific failure patterns. The efficiency is assessed in terms of work, time, and communication complexity depending on the specific model of computation.

In this paper we initiate the study of the Do-All problem under *Byzantine processor failures* [21] that model arbitrary processor malfunction. We consider synchronous processors that communicate by exchanging messages. We assume that the execution of a single task takes bounded constant time, modeled as one computation step for any processor. The tasks can be performed in any order and multiple executions of the same task do not affect the outcome of the computation. We evaluate algorithms according to the number of computation steps taken by the processors during the computation, i.e., the *available processor steps* or *work* measure of Kanellakis and Shvartsman [17], and according to their *communication cost* that counts the number of point-to-point messages sent by the processors during the computation. The work and messages of a faulty processor are counted only until it fails to follow the algorithm being executed.

The available processor steps measure is a direct generalization of the $\mathsf{processor} \times \mathsf{time}$ product, a standard complexity measure in parallel computing. Both complexity measures account for all steps of participating processors, including any idling steps. This is especially relevant in the context of *fast algorithms* where the goal is to complete the required work as efficiently as possible and as fast as possible (which is the natural concern in practical applications such as factorization for public-key cryptanalysis). Hence, by "forcing" all non-faulty processors to work at every step (and not allowing the processors to idle for

free until the computation is complete) we employ the full available parallelism. In the study of the Do-All problem, this enable us to extract and identify the trade-offs between efficiency and fault-tolerance in the most general case, where processors must work until all tasks are performed, and despite the failures in the system components. In this paper our goal is to obtain fast algorithms, and hence, using the available processor steps measure to evaluate the efficiency of our solutions and lower bounds, is a natural choice. Evaluating our solutions in terms of message complexity is also important, as being fast is not our only goal and it is important to manage the communication efficiency as well.

Prior work. The Do-All problem, its shared-memory version, the *Write-All* problem, and its partitionable networks version, the Omni-Do problem, have been studied under various failure assumptions. However, this problem has not been studied under Byzantine processor failures. Prior work on Do-All dealt with processor stop-failures (e.g., [17,8,10,14,6]), with processor stop-failures and restarts (e.g., [18,4]), with networks prone to partition (e.g., [9,22,13]), and with processor delays (e.g., [23,3,2,5,16]).

The model of Byzantine processor failures was introduced by Lamport et al. [21] in the context of the consensus problem (a set of processors must agree on a common value). Assuming that the number of faulty processors f is fixed and known in advance, (among other results) they gave a lower bound of $3f + 1$ for the number of synchronous processors required for consensus. They also presented a synchronous consensus protocol that works in $f + 1$ rounds, with $n > 3f + 1$, but exponential communication (number of messages).

Contributions. This paper presents the first results for the Do-All problem for synchronous message-passing processors prone to Byzantine failures. Let m be the number of tasks to be executed, and n the number of processors, of which up to f can fail. Note that the Do-All problem can be trivially solved with $\Theta(mn)$ work by having each processor perform all the tasks. Thus the goal is to seek solutions with $o(mn)$ work, or to show that no such solutions are possible.

We study this problem in several settings. We consider (a) the case where the maximum number of faulty processors f is known a priori, (b) the case where f is not known, (c) the case where a task execution can be verified (without re-executing the task), and (d) the case where task executions cannot be verified. Fig. 1 summarizes the results obtained in this paper. For these results we assume $n \leqslant m$ (this is the most interesting case, when the number of processors n does not exceed the number of tasks m). In the figure, v, $1 \leqslant v \leqslant m$, is the number of tasks whose completion status can be verified by one processor in one step. Here $\phi \leqslant f$ is the actual number of processors that fail in a computation of interest (f is the upper bound on the number of processors that may fail; of course a smaller number of processor may actually fail in a specific execution). For brevity, we define $\Lambda_{n,\phi}$ as follows: $\Lambda_{n,\phi} = \log(n/\phi)$ when $\phi \leqslant n/\log n$ and $\Lambda_{n,\phi} = \log\log n$ when $n/\log n < \phi < n$. Where the upper bounds on the communication complexity are not given, the work bounds are obtained without communication. Finally, we use Θ notation to specify upper and lower bounds in conjunction with work defined to be the minimum, over all algorithms, of the maximum work caused by all adversaries.

<table>
<tr><td></td><td>No verification</td><td colspan="2">Verification: v tasks can be verified in one step</td></tr>
<tr><td></td><td></td><td>$f = \Theta(n)$</td><td>$f = o(n)$</td></tr>
<tr><td rowspan="2">Known
f</td><td rowspan="2">Work
$\Theta(m(f+1))$</td><td rowspan="2">Work
$\Theta\left(m + \frac{mn}{v} + \frac{n \log n}{\Lambda_{n,\phi}}\right)$</td><td>Work
$\Omega\left(m + \frac{mf}{v} + \frac{n \log n}{\Lambda_{n,\phi}}\right)$</td></tr>
<tr><td>$O\left(m + \frac{mf}{v} + n(1 + \frac{f}{v}) \cdot \min\{\phi + 1, \log n\}\right)$
with communication
$O(n(f+1) \cdot \min\{\phi + 1, \log n\})$</td></tr>
<tr><td>Unknown
f</td><td>Work
$\Theta(mn)$</td><td colspan="2">Work
$\Theta\left(m + \frac{mn}{v} + \frac{n \log n}{\Lambda_{n,\phi}}\right)$</td></tr>
</table>

Fig. 1. Summary of the results.

Among the different assumptions considered, the verifiability of tasks is possibly the least common. The assumption is that in a system with verification processors can check whether a task has been executed (up to v tasks can be verified in one step). Depending on the kind of tasks we are considering, this capability could be provided by several means. For instance, the tasks could be computational problems such that solving them is significantly more costly than checking whether a given candidate solution is correct. (Examples of these problems are sorting a list or factoring a large number.) In this case, a processor can verify that the task has been done if it has a correct solution. The problem of distributing these solutions to the processors can be solved, for instance, with a reliable stable storage holding a database of solutions, which upon request delivers the solutions to given tasks and which does not accept incorrect solutions (it verifies them before adding them to the database). Another possibility would be having each processor reliably broadcasting the obtained solution after each task execution. Note that in both cases it is simple to enforce that in the same time step two correct processors either both find a task done or undone. In the rest of the paper we do not consider the specific verification methods and we abstract the cost of verification in terms of the parameter v. Furthermore, we do not count the messages (if any) involved in the verification, since this is dependent on the particular verification methods and need not be a function of the number of verifications.

We remark that in the preliminary version of this work [11], some results were incorrectly stated. Specifically, for the cases of unknown f with verification and of known $f = \Theta(n)$ with verification, the upper bound on work was given as $O(mn/v)$, for the full ranges $n \leqslant m$ and $v \leqslant m$. Here we show that the above upper bound holds precisely when $v = O(n)$ and $m/v = \Omega(\log n/\Lambda_{n,\phi})$.

More interestingly, in [11] the lower bound on work for the same cases was given as $\Omega(mn/v)$. Here we show the stronger lower bound on work of $\Omega(m+mn/v+n \log n/\Lambda_{n,\phi})$. Furthermore, in [11], for the case of known $f = o(n)$ with verification, the lower bound on work was shown to be $\Omega(m(f+1)/v)$. Here we show the stronger lower bound on work of $\Omega(m + mf/v + n \log n/\Lambda_{n,\phi})$.

Document structure. The paper is organized as follows. In Section 2 we define the model of computation, the Do-All problem, and the complexity measures. In Section 3 we present our results when the task executions cannot be verified, first for the case when the maximum number of faulty processors f is known (Section 3.1) and then for the case when f is unknown

(Section 3.2). In Section 4 we present our results when the task executions can be verified, first when f is known (Section 4.1) and then when f is unknown (Section 4.2). We conclude in Section 5.

2. Model of computation

We start by defining the system model, the abstract problem of performing a collection of tasks in a distributed environment with Byzantine failures, and the complexity measures of interest.

Distributed setting. We consider a distributed system consisting of n synchronous message-passing processors; each processor has a unique identifier (PID) from the set $[n] = \{1, 2, \ldots, n\}$. We assume that n is fixed and is known to all processors.

Tasks. We define a *task* to be a computation that can be performed by any processor in one step. An execution of any task does not depend on the executions of other tasks. The tasks are *idempotent*, i.e., executing a task many times or concurrently with other tasks has the same effect as executing the task once by itself. Each task has a unique identifier (TID) from the set $[m] = \{1, 2, \ldots, m\}$. We assume that all m tasks are initially known to all processors.

We consider the setting where a task execution can be *verified* without re-executing the task and the setting where a task execution cannot be verified. When verification is possible, we assume that up to v tasks, $1 \leqslant v \leqslant m$, can be verified by a processor in one step. Because the setting is synchronous, we assume that if the same task is verified by several processors in the same step (see below), then either all processors find the task done or all of them find the task undone. As we mentioned previously, the verification could be done with different techniques, and our model is a simple abstraction of any of these techniques.

Synchrony and time. We consider the synchronous model where the processors proceed in lock-step, and assume that in each synchronous step a processor can: (1) execute a task or verify up to v tasks (when verification can be done), (2) send messages to other processors and receive the messages sent to it by other processors in the same step, and (3) perform a constant-time local computation. We measure *time complexity* in terms of the synchronous parallel steps.

Communication. Processors communicate by sending point-to-point messages. The underlying communication network is assumed to be fully connected, that is, any processor can send messages to any other processor. We assume that messages are neither lost nor corrupted in transit, and that messages contain $O(\max\{m, n\})$ bits. Messages sent in one step of the computation are received in the same step.

Model of failures. We consider *Byzantine processor failures* [21]. We assume that a faulty processor can behave arbitrarily (do nothing, do something not directed by its protocol, send arbitrary messages, or behave normally). A faulty processor controls only its own messages and its own actions, and it cannot control other processors' messages and actions. In

particular, a faulty processor cannot corrupt another processor's state, modify/replace another processor's messages, or impersonate other processors (i.e., create and send messages that appear to have been sent by another processor). A faulty processor cannot "undo" a task that was previously executed.

We let an omniscient *adversary* impose Byzantine failures on the system. We use the notion of a *failure pattern* to describe the occurence of Byzantine failures caused by the adversary in a given computation. Syntactically, a failure pattern F is a set of pairs (p, t), where t is the first time step of the computation where the adversary forces processor $p \in [n]$ to behave differently from what is prescribed by the algorithm for processor p. We assume that the adversary has full knowledge of the actions and decisions taken by the algorithm before step t (i.e., the adversary has full knowledge of the history of the computation).

When a computation occurs in the presence of a failure pattern F, we say that processor $p \in [n]$ *survives* step i if F does not contain a pair (p, t) such that $t \leqslant i$. We say that a processor p *fails* in F, if there exists a pair (p, t) in F, for some t. For a failure pattern F we define its *size* to be $\phi = |F|$, i.e., it is the number of processors that fail in F (ϕ can be 0).

A *failure model* $\mathcal{F}$ is the set of all failure patterns that a given adversary can force. For the purpose of this paper we consider failure models $\mathcal{F}_f$, where $f < n$, that contain all possible failure patterns of size at most f. In this work we analyze the case where the parameter f is known to the algorithms, and the case where f is unknown.

The Do-All **problem.** We define the Do-All problem as follows:

Do-All: Given a set of m tasks, perform all tasks using n processors, in the presence of any failure pattern F in the given failure model $\mathcal{F}$.

The Do-All problem is considered to be solved when all m tasks are performed and at least one non-faulty processor is aware of this.

Work and message complexity. We are interested in studying the complexity of the Do-All problem measured as *work* (cf. [17,10,8]). We assume that a single step of a processor corresponds to a unit of work (recall that a single task can be performed in a single step). When task verification is allowed, we assume that up to v tasks can be verified in a single step. Thus performing a task or verifying v tasks corresponds to a unit of work. Our definition of *work complexity* is based on the *available processor steps* measure [17]. For a computation subject to a failure pattern $F \in \mathcal{F}_f$, denote by $P_i(F)$ the number of processors that survive step i of the computation.

Definition 2.1 (*Work complexity*). Given a problem of size m and an n-processor algorithm that solves the problem in the failure model $\mathcal{F}_f$, if the algorithm solves the problem for a failure pattern $F \in \mathcal{F}_f$ by step $\tau(F)$, then the work complexity S of the algorithm is

$$S = S_{m,n,f} = \max_{F \in \mathcal{F}_f} \left\{ \sum_{i=1}^{\tau(F)} P_i(F) \right\}.$$

We also evaluate the efficiency of message-passing algorithms in terms of their *message complexity*. For a computation subject to a failure pattern $F \in \mathcal{F}_f$, denote by $M_i(F)$ the

number of point-to-point messages sent at step i of the computation by the $P_i(F)$ processors that survive that step.

Definition 2.2 (*Message complexity*). Given a problem of size m and an n-processor algorithm that solves the problem in the failure model $\mathcal{F}_f$, if the algorithm solves the problem for a failure pattern $F \in \mathcal{F}_f$ by step $\tau(F)$, then the message complexity M of the algorithm is

$$M = M_{m,n,f} = \max_{F \in \mathcal{F}_f} \left\{ \sum_{i=1}^{\tau(F)} M_i(F) \right\}.$$

Note that when processors communicate using broadcasts or multicasts, each broadcast/multicast is counted as the number of point-to-point messages from senders to receivers. As we mentioned in the previous section, we do not count as part of the message complexity of the algorithm the messages used to verify tasks in the models with verifications.

In the rest of the paper we assume that, initially, the number of processors n is no more than the number of tasks m; these are the scenarios motivated by typical applications. Analysis for cases when $n > m$ follows *mutatis mutandis*.

3. Doing-all when task execution is not verifiable

We first consider the setting where a processor cannot verify whether or not a task was performed. Thus a faulty processor can "lie" about doing a task without any other processor being able to detect it.

3.1. The maximum number of faulty processors is known

We assume here that the upper bound f on the number of processors that can fail is known a priori; of course the set of processors that may actually fail in any given execution is not known. We first present lower bounds for this setting.

Theorem 3.1. *Any fault-free execution of an algorithm that solves the* Do-All *problem in the failure model $\mathcal{F}_f$ with f known, takes at least $\lceil \frac{m(f+1)}{n} \rceil$ steps.*

Proof. By way of contradiction, assume that there is an algorithm A that solves the Do-All problem for all failure patterns of size at most f, and that it has some failure-free execution R that solves the problem in $s < \lceil (m(f+1)/n) \rceil$ steps. Then, in R there is a task z that has been performed by less than $f+1$ processors, since $\lfloor \frac{sn}{m} \rfloor \leqslant \lfloor \frac{(\lceil m(f+1)/n \rceil - 1)n}{m} \rfloor < f+1$.

Now construct an execution R' of A that behaves exactly like R except that in the first s steps each processor that is supposed to execute task z is in fact faulty and does not execute z. Since z is executed by less than $f+1$ processors, z is not executed. Since verification is not available, no correct processor in R' can distinguish R from R', hence R' stops after s steps and the problem is not solved (since at least one task was not performed), a contradiction. □

Processor p, $1 \leqslant p \leqslant n$ **does:**
1 for $k_p = 1$ to $\lceil \frac{m(f+1)}{n} \rceil$ do
2 execute task $((\lceil \frac{mp}{n} \rceil + k_p) \bmod m) + 1$

Fig. 2. Algorithm *Cover*. The code is for processor p.

Corollary 3.2. *Any fault-free execution of an algorithm that solves the* Do-All *problem in the failure model $\mathcal{F}_f$ with f known, has work at least $\lceil m(f+1)/n \rceil n$.*

We now present algorithm *Cover* that solves Do-All in the case where *f* is known and task execution cannot be verified. The algorithm is simple: each task is performed by $f+1$ processors. Since there can be at most *f* faulty processors, this guarantees that each task is performed at least once. This implies the correctness of the algorithm. The pseudocode of the algorithm is given in Fig. 2. We now show that algorithm *Cover* is optimal.

Theorem 3.3. *Algorithm Cover solves the* Do-All *problem in the failure model $\mathcal{F}_f$ with f known, in optimal number of steps $\lceil m(f+1)/n \rceil$ and work $\lceil m(f+1)/n \rceil n$, without any communication.*

Proof. The proof follows from the fact that each task is executed by at least $f+1$ different processors. Since at most *f* processors are faulty, at least one correct processor executes the task.

For simplicity we will remove the modular algebra (see Fig. 2) for both processor and task indices. We do this by assuming that any task number z, $z < 1$, is in fact the task number $z+m$, any task number $z > m$ is in fact the task number $z-m$, and any processor p, with $p < 1$, is in fact processor $p+n$.

Let us consider the tasks between $\lceil \frac{mp}{n} \rceil + 2$ and $\lceil \frac{m(p+1)}{n} \rceil + 1$. We show that these tasks are executed by processors $p-f$ to p. For that, it is enough to show that the last task executed by processor $p-f$ is at least task number $\lceil m(p+1)/n \rceil + 1$. This can be simply observed, since $\lceil m(p-f)/n \rceil + \lceil m(f+1)/n \rceil + 1 \geqslant \lceil m(p+1)/n \rceil + 1$, from the fact that $\lceil x \rceil + \lceil y \rceil \geqslant \lceil x+y \rceil$. □

It is worth observing that algorithm *Cover* is work-optimal and time-optimal even though no communication took place. This shows that in this setting communication does not help obtaining better performance.

3.2. The maximum number of faulty processors is unknown

Now we consider the case where the upper bound *f* is not known, i.e., all that is known is that $f < n$. In this setting we observe that no algorithm can do better than having each processor perform each task, as shown in the following theorem.

Theorem 3.4. *Any fault-free execution of an algorithm that solves the* Do-All *problem in the failure model $\mathcal{F}_{n-1}$ takes at least m steps and incurs at least $m \cdot n$ work.*

This is an immediate corollary of the above discussion. In summary, it is not very interesting to study fault-tolerant computation in this model:

Corollary 3.5. *When f is unknown and the task execution cannot be verified, the trivial algorithm in which each processor executes all the tasks is optimal.*

4. Doing-all when task execution is verifiable

Given the pessimistic results in Section 3 regarding our ability to solve Do-All efficiently, we study the problem under a new assumption. We assume that there is a way for a processor to verify whether a task has been done or not (without executing the task). The verification mechanism reinforces the ability of correct processors to detect faulty processors: if a faulty processor "lies" about having done a task, a correct processor can detect this by separately verifying the execution of the task. Here we assume that in one step any processor can verify up to v tasks, where $1 \leqslant v \leqslant m$.

4.1. The maximum number of faulty processors is known

As before, we first consider the case where the upper bound f on the number of faulty processors is known. We first show lower bounds on steps and work required by any Do-All algorithm in this case (Section 4.1.1). Then we present an algorithm, called *Minority*, designed to efficiently solve Do-All when $f \geqslant n/2$ (Section 4.1.2). Next we present algorithm *Majority* that is designed to efficiently solve Do-All when $f < n/2$ (Section 4.1.3). Finally, we combine algorithms *Minority* and *Majority*, yielding an algorithm, called *Complete*, that efficiently solves Do-All for the whole range of f (Section 4.1.4). The complexity of algorithm *Complete* depends on f and comes close to matching the corresponding lower bound.

4.1.1. Lower bounds

We now present lower bounds on time steps and work for any execution of an algorithm that solves the Do-All problem with verification and known f. The first result is a bound on work that follows directly from the analogous result shown in [14] for the fail-stop model [25] (a processor may crash at any moment during the computation and once crashed it does not restart). Recall that we define $\Lambda_{n,\phi}$ as follows: $\Lambda_{n,\phi} = \log(\frac{n}{\phi})$ when $\phi \leqslant n/\log n$, and $\Lambda_{n,\phi} = \log\log n$ when $n/\log n < \phi < n$.

Lemma 4.1. *Any execution of an algorithm that solves the Do-All problem in the failure model $\mathcal{F}_f$ with f known, in the presence of $\phi \leqslant f$ Byzantine failures, requires work $\Omega(m + n\log n/\Lambda_{n,\phi})$.*

Proof. Theorem 2 in [14] gives a lower bound on the amount of work any algorithm that solves the Do-All problem requires. The mentioned theorem assumes the fail-stop model, and the existence of an oracle that gives information about termination and that balances the undone tasks among the correct processors. Implicitly, the oracle can verify the execution of

up to m tasks in constant time. The theorem shows that, just in executing tasks, any execution with ϕ failures of an algorithm that solves Do-All (for $n \leqslant m$) in this model requires work $\Omega(m + n \log n / \Lambda_{n,\phi})$. Since crashes are a special case of Byzantine failures, the lower bound applies to our model as well. □

We now present a lower bound on the steps of any algorithm that solves the Do-All problem.

Lemma 4.2. *Any fault-free execution of an algorithm that solves the* Do-All *problem in the failure mode $\mathcal{F}_f$ with f known and with task verification, takes at least $\lceil m(f+v)/nv \rceil$ steps.*

Proof. By way of contradiction, assume that there is an algorithm A that solves the Do-All problem with verification for all failure patterns of length at most f and it has some failure-free execution R that solves the problem in $s < \lceil m(f+v)/nv \rceil$ steps (since s is an integer, we can drop the ceiling: $s < m(f+v)/nv$). The work in this execution is $s \cdot n$. Note that in these steps each task has been executed at least once. Counting just one task execution, m units of work have been spent on executing the tasks. The remaining work is $sn - m$, and each work unit can be used to either perform a task or to verify v of them. Then there is a task z that, in addition to having been executed once, has been "looked at" (executed or verified) at most $f - 1$ more times, since

$$\left\lfloor \frac{(sn-m)v}{m} \right\rfloor < \left\lfloor \frac{(\frac{m(f+v)}{nv}n - m)v}{m} \right\rfloor = \left\lfloor \left(\frac{f+v}{v} - 1 \right) v \right\rfloor = f$$

(by the pigeonhole principle). Thus task z has been "looked at" at most f times.

Now construct an execution R' of A that behaves exactly like R except that in the first s steps each processor that is supposed to execute task z is in fact faulty and does not execute it, and every processor that is supposed to verify z is also faulty and behaves as if z was executed. Then, no correct processor in R' can distinguish R from R', hence R' stops after s steps and the problem is not solved (since at least one task was not performed), a contradiction. □

The above lemma leads to the following result.

Theorem 4.3. *Any fault-free execution of an algorithm that solves the Do-All problem in the failure model $\mathcal{F}_f$ with f known and with task verification, requires work $\lceil m(f+v)/nv \rceil \cdot n$.*

Proof. Using Lemma 4.2 and the fact that none of the n processors fail, we compute the work of any algorithm as $\lceil m(f+v)/nv \rceil \cdot n$. □

From the above we obtain the following lower bound result.

Theorem 4.4. *Any algorithm that solves the Do-All problem in the failure model $\mathcal{F}_f$ with f known, in the presence of $\phi \leqslant f$ Byzantine failures, and with task verification, incurs work $\Omega(m + mf/v + n \log n / \Lambda_{n,\phi})$.*

Minority(p, P, T, ψ)**:**
1 while $T \neq \emptyset$ and $\psi > 0$ do
2 execute one task allocated to p as a function of p, P, and T
3 $\Phi \leftarrow \emptyset$
4 $C \leftarrow$ tasks allocated to the processors in P, as a list of $\lceil \frac{\min\{|P|,|T|\}}{v} \rceil$ sets of at most v tasks each
5 for $l = 1$ to $\lceil \frac{\min\{|P|,|T|\}}{v} \rceil$ do
6 verify the tasks in the lth set $C[l]$
7 $\Phi \leftarrow \Phi \cup \{k :$ task $z \in C[l]$ was allocated to processor k and was not done$\}$
8 end for
9 $P \leftarrow P \setminus \Phi$
10 $T \leftarrow T \setminus \{z : z$ was allocated to some $k \in P\}$
11 $\psi \leftarrow \psi - |\Phi|$
12 end while
13 execute up to $\lceil |T|/|P| \rceil$ tasks allocated to p as a function of p, P, and T

Fig. 3. Algorithm for the case $f \geqslant n/2$. The code is for processor p. The call to the procedure is made with $P = [n]$, $T = [m]$, and $\psi = f$.

Proof. It follows directly from Lemma 4.1 and Theorem 4.3. □

4.1.2. **Algorithm** *Minority*

Now we present algorithm *Minority* that is designed to solve Do-All in the case when at most half of the processors are guaranteed not to fail, i.e., $f \geqslant n/2$. Algorithm *Minority* is detailed in Fig. 3. The code is given for a generic processor $p \in [n]$.

As can be seen in Fig. 3, the main body of the algorithm is formed by a while loop. Within the loop the variables P, T, and ψ are updated so they always hold the current set of the processors assumed to be correct, the tasks whose completion status is unknown, and the number of processors that can still fail, respectively. The iterations of the while loop are executed synchronously by every correct processor. An important correctness condition of the algorithm is that every correct processor has the same value in these variables at the beginning of each loop iteration (that is why we do not index the variables with the processor's id). The exit conditions of the loop are that there is no remaining work or no remaining processor is faulty. If the latter condition holds, then the remaining tasks are evenly distributed among the remaining processors in P, so that every tasks is assigned to at least one processor, and the problem is solved.

Consider an execution of algorithm *Minority*. Let k be the number of iterations of the while loop in this execution. The iterations are numbered starting with 1. We denote by P_i, T_i, and ψ_i the values of the sets P and T, and the variable ψ, respectively, at the *end* of iteration i. We also use P_0, T_0, and ψ_0 to denote the initial values of P, T, and ψ, respectively. To abbreviate, we use $n_i = |P_i|$ and $m_i = |T_i|$.

For an iteration i of the loop, each processor first chooses one of the tasks in T_{i-1} deterministically with an allocating function of p, P_{i-1}, and T_{i-1}. The allocating function is known to every processor and must ensure that, if $m_{i-1} \geqslant n_{i-1}$, different processors in P_{i-1} choose different tasks in T_{i-1}. It must also ensure that if $m_{i-1} < n_{i-1}$, each task is assigned to at least $\lfloor n_{i-1}/m_{i-1} \rfloor$ and at most $\lceil n_{i-1}/m_{i-1} \rceil$ processors. One possible allocating function is one that (once the processors in P_{i-1} are indexed from 1 to n_{i-1}

```
Oracle(p):
1  while there are undone tasks do
2      allocate an undone task t to p
3      execute task t
4  end while
```

Fig. 4. Oracle-based Do-All algorithm under the fail-stop model. The code is for processor p.

and the tasks in T_{i-1} are indexed from 1 to m_{i-1}) assigns to the qth processor in P_{i-1} the $(((q-1) \bmod m_{i-1}) + 1)$st task in T_{i-1}. After executing this task, the processor verifies the execution of all the tasks allocated to processors in P_{i-1} to identify faulty processors. The identities of the newly discovered faulty processors are stored in the set Φ. With this information it updates the sets T_{i-1} and P_{i-1} and the value ψ_{i-1} and obtains T_i, P_i and ψ_i, respectively. The list of sets C is the same for each processor. Then, according to the description of the algorithm all processors verify the tasks allocated to a subset of correct processors simultaneously, either finding each of them done or undone. This guarantees that the sets Φ are the same in all correct processors.

The correctness of algorithm *Minority* can be shown by induction on the number of iterations of the while loop. The induction claims that at the beginning of iteration $i > 0$ all correct processors have the same value of the variables P_{i-1}, T_{i-1}, and ψ_{i-1}, and that $|T_{i-1}| \leqslant m - i + 1$. Observe that initially all processors have the same P_0, T_0 and ψ_0, and that $|T_0| = m$, which covers the base case. The induction then has to show that if the correct processors begin an iteration i with the same P_{i-1}, T_{i-1} and ψ_{i-1}, then at the end of this iteration all correct processors have the same P_i, T_i, and ψ_i, and at least one new task has been done in the iteration. The first part follows from the fact that all correct processes end up with the same set Φ of failed processes. The second follows from the fact that at least one processor is correct. Then, termination is guaranteed (with all tasks being completed) by the fact that at least one processor is correct ($f < n$), by the fact that after at most m iterations all correct processors will exit the while loop, by the exit conditions of the while loop, and by "line 13" of the code of the algorithm. We leave the details of the termination to the reader.

We now assess the efficiency of algorithm *Minority*. We denote by Φ_i the value of set Φ at the *end* of iteration i of the loop, and we use $\phi_i = |\Phi_i|$. Recall that ϕ denotes the number of failed processors in a given execution.

Towards the analysis, we first present the algorithm *Oracle*, shown in Fig. 4, which uses an oracle to solve the synchronous Do-All problem under the fail-stop processor model.

In algorithm *Oracle*, the oracle is queried in each iteration to determine whether there are still undone tasks. The oracle can detect the processors that crashed during an iteration and whether a task has been performed or not by the end of the iteration (if all processors assigned to a task have crashed, then the task has not been performed). If there is at least one undone task by the end of the iteration, then the oracle is queried to allocate undone tasks to the uncrashed processors. The allocation satisfies that the undone tasks are evenly distributed among the uncrashed processors. In fact, we assume here that the function that allocates in each iteration an undone task to processor p (for each processor p) in line 2 is the same

used in algorithm *Minority*. Hence, the difference between algorithm *Oracle* and algorithm *Minority* is that in algorithm *Minority* the task execution verification is performed by the processors to detect faulty processors and undone tasks, as opposed to algorithm *Oracle* where the task execution verification is performed by the oracle. Algorithm *Oracle* is a rewriting of the oracle-based algorithm presented by Georgiou et al. [14]. Assuming that the queries to the oracle can be done in constant time, they showed that in an execution with no more than ϕ crashes the algorithm *Oracle* requires at most work $\mathrm{O}\left(m + n \log n / \Lambda_{n,\phi}\right)$. We will use this result to show Lemma 4.5 for algorithm *Minority*.

Lemma 4.5. *Given an execution of algorithm Minority with ϕ failures and where the while loop consists of k iterations, then $\sum_{i=1}^{k} n_i = \mathrm{O}\left(m + n \log n / \Lambda_{n,\phi}\right)$.*

Proof. Consider an execution of the algorithm *Minority* with ϕ failures, and let k be the number of iterations of the while loop. We want to bound the sum $\sum_{i=1}^{k} n_i$. For that, let us consider the execution of algorithm *Oracle* in which after the allocation at line 2 and before the task execution at line 3 in each iteration $i \in \{1, \ldots, k\}$ the processors in Φ_i, and only those, crash. Then, since the same allocation function is used in the executions of *Minority* and *Oracle*, it follows by induction on i that in the execution of algorithm *Oracle*, at the beginning of iteration i the set of correct processors is P_{i-1} and the set of undone tasks is T_{i-1}, and at the end of the iteration the set of correct processors is P_i and the set of undone tasks is T_i. Observe that for algorithm *Oracle*, when the oracle queries can be done in constant time, we have that the work of iteration i, denoted s_i, is a constant multiple of the number of correct processors n_i. Hence, if we denote by S_k the work of the k first iterations of *Oracle*, we have that

$$S_k = \sum_{i=1}^{k} s_i \geqslant \sum_{i=1}^{k} n_i . \tag{1}$$

Now, since the number of failures in the execution of *Minority* is ϕ, if we assume that no processor crashes after iteration k in the execution of *Oracle* we have that the number of failures in this execution is $\sum_{i=1}^{k} \phi_i \leqslant \phi$. Hence, from the result of Georgiou et al. [14] mentioned above, we have that

$$S_k = \mathrm{O}\left(m + n \log n / \Lambda_{n,\phi}\right). \tag{2}$$

The thesis of the lemma follows from Eqs. (1) and (2). □

We now state and prove the work complexity of algorithm *Minority*.

Lemma 4.6. *Any execution of algorithm Minority has work $S = \mathrm{O}(m + mn/v + n \log n / \Lambda_{n,\phi})$.*

Proof. We begin by computing the work incurred in the while loop. We break the analysis into two parts. In the first part we consider only the iterations i of the while loop where initially the number of remaining tasks is at least as large as the number of remaining processors, i.e., $m_{i-1} \geqslant n_{i-1}$, and we compute the work incurred in these iterations. In the

second part we consider only the iterations i where $m_{i-1} < n_{i-1}$ and we compute the work incurred in such iterations.

(1) *Iterations i with $m_{i-1} \geqslant n_{i-1}$.* In these iterations no task is done twice by correct processors. Hence, at most m tasks are done in these iterations. For each task done, no more than $\lceil n/v \rceil < n/v + 1$ verification steps are taken. Hence, the total work incurred in these iterations is $S_1 = \mathrm{O}(m + mn/v)$.

(2) *Iterations i with $m_{i-1} < n_{i-1}$.* Let us assume there are r such iterations out of a total of k iterations ($r \leqslant k$), with indices $\ell^{(1)}$ to $\ell^{(r)}$, and $1 \leqslant \ell^{(1)} < \ell^{(2)} < \cdots < \ell^{(r)} \leqslant k$. In iteration $\ell^{(i)}$, there are initially $n_{\ell^{(i)}-1}$ processors and $m_{\ell^{(i)}-1}$ tasks, with $m_{\ell^{(i)}-1} < n_{\ell^{(i)}-1}$. In this iteration each (correct) processor performs a task and verifies $\min\{n_{\ell^{(i)}-1}, m_{\ell^{(i)}-1}\} = m_{\ell^{(i)}-1}$ tasks. Hence, the total work incurred in all r iterations is

$$S_2 = \sum_{i=1}^{r} n_{\ell^{(i)}} \left(1 + \left\lceil \frac{m_{\ell^{(i)}-1}}{v} \right\rceil\right) < 2 \sum_{i=1}^{r} n_{\ell^{(i)}} + \frac{1}{v} \sum_{i=1}^{r} n_{\ell^{(i)}} m_{\ell^{(i)}-1}.$$

The first sum is bounded by using Lemma 4.5, since

$$\sum_{i=1}^{r} n_{\ell^{(i)}} \leqslant \sum_{i=1}^{k} n_i = \mathrm{O}\left(m + n \log n / \Lambda_{n,\phi}\right).$$

To bound the second sum, we bound first the value of $m_{\ell^{(i)}}$ using the fact that, in iteration $\ell^{(i)}$, each task is assigned to at most $\lceil n_{\ell^{(i)}-1}/m_{\ell^{(i)}-1} \rceil$ processors,

$$m_{\ell^{(i)}} \leqslant m_{\ell^{(i)}-1} - \frac{n_{\ell^{(i)}}}{\lceil n_{\ell^{(i)}-1}/m_{\ell^{(i)}-1} \rceil} < m_{\ell^{(i)}-1} - \frac{n_{\ell^{(i)}} m_{\ell^{(i)}-1}}{m_{\ell^{(i)}-1} + n_{\ell^{(i)}-1}}.$$

Then, since $m_{\ell^{(i)}-1} < n_{\ell^{(i)}-1}$, we have

$$n_{\ell^{(i)}} m_{\ell^{(i)}-1} < 2\, n_{\ell^{(i)}-1}\left(m_{\ell^{(i)}-1} - m_{\ell^{(i)}}\right). \tag{3}$$

Then, the second sum can be bounded as follows:

$$\begin{aligned}
\sum_{i=1}^{r} n_{\ell^{(i)}} m_{\ell^{(i)}-1} &< \sum_{i=1}^{r} 2 n_{\ell^{(i)}-1}\left(m_{\ell^{(i)}-1} - m_{\ell^{(i)}}\right) \\
&\leqslant 2\left(n_{\ell^{(1)}-1} m_{\ell^{(1)}-1} + \sum_{i=2}^{r} n_{\ell^{(i)}-1} m_{\ell^{(i)}-1} - \sum_{i=1}^{r-1} n_{\ell^{(i)}-1} m_{\ell^{(i)}}\right) \\
&\leqslant 2\left(n_{\ell^{(1)}-1} m_{\ell^{(1)}-1} + \sum_{i=1}^{r-1} m_{\ell^{(i)}}\left(n_{\ell^{(i+1)}-1} - n_{\ell^{(i)}-1}\right)\right) \\
&\leqslant 2 n_{\ell^{(1)}-1} m_{\ell^{(1)}-1} \\
&\leqslant 2mn.
\end{aligned}$$

The first inequality follows from Eq. (3), the third inequality follows from the fact that $m_{\ell^{(i)}-1} \leqslant m_{\ell^{(i-1)}}$, and the fourth inequality follows from the facts that $\ell^{(i+1)} - 1 > \ell^{(i)} - 1$ and that $n_i \leqslant n_j$ when $i > j$.

Then, we have that the work incurred in these iterations is $S_2 = \mathrm{O}(m + mn/v + n \log n/\Lambda_{n,\phi})$.

We now compute the work incurred after the exit conditions are satisfied, say at the end of iteration k. If $T_k = \emptyset$ then each processor takes at most one step before halting for the

```
Majority(p, P, T, ψ):
1  while |T| > m/n and ψ > 0 do
2      Do_Work_and_Verify(p, P, T, ψ, Φ)
3      Checkpoint(p, P, ψ, Φ)
4      P ← P \ Φ
5      T ← T \ {z : z was allocated to some k ∈ P}
6      ψ ← ψ − |Φ|
7  end while
8  if ψ = 0 then
9      execute ⌈|T|/|P|⌉ tasks allocated to p as a function of p, P, and T
10 else
11     execute all the tasks in T
12 end if
```

Fig. 5. Algorithm for the case $f < n/2$. The code is for processor p. The call parameters are $P = [n]$, $T = [m]$, and $\psi = f$.

total of $O(n)$ work. Otherwise, at most $n\lceil m/n\rceil < m + n \leqslant 2m$ work is done. Hence this work is $S_3 = O(m)$.

Therefore, the total work is $S = S_1 + S_2 + S_3 = O\left(m + mn/v + n \log n/\Lambda_{n,\phi}\right)$. □

Note that the work complexity of the algorithm is asymptotically optimal as long as $f = \Omega(n)$. It is worth observing that algorithm *Minority* is asymptotically optimal even though it does not use communication. This shows that for relatively large number of failures communication cannot improve work complexity (asymptotically).

Remark 4.1. In the conference version of this paper [11], the bound on the work for *Minority* was imprecisely given as $O(mn/v)$, for any $n \leqslant m$ and $v \leqslant m$. As it can be observed from Lemma 4.6, this bound is valid only as long as $v = O(n)$ and $m/v = \Omega(\log n/\Lambda_{n,\phi})$.

4.1.3. **Algorithm** *Majority*

We now present algorithm *Majority* that is designed to efficiently solve Do-All in the case where the majority of the processors does not fail, i.e., $f < n/2$. At a high level algorithm *Majority* proceeds as follows. Each nonfaulty processor is given a set of tasks to be done and a set of processors whose tasks it has to verify. The processor executes its tasks and verifies the tasks of its set of processors, detecting faulty processors. Then a check-pointing algorithm is executed in which all nonfaulty processors agree on a set of processors identified as faulty in this stage, and update their information of completed tasks and non-faulty processors accordingly. Algorithm *Majority* is detailed in Fig. 5. The code is given for a processor $p \in [n]$.

As in algorithm *Minority*, the parameters of algorithm *Majority* are the identifier p of the invoking processor, the set of processors P that have not been identified as faulty, the set of tasks T that may still need to be completed, and the maximum number ψ of processors in set P that can be faulty. We adopt the parameter notations we used for an iteration of the while loop of algorithm *Minority* to the parameters of algorithm *Majority*. Specifically, for an iteration i, we let P_i, T_i and ψ_i denote the values of P, T, and ψ, respectively, at the *end* of iteration i. Then, $n_i = |P_i|$ and $m_i = |T_i|$. Finally, $n_0 = n$ and $m_0 = m$.

The iterations of the while loop of *Majority* in all the correct processors work synchronously, i.e., the *i*th iteration starts at exactly the same step in each correct processor. An important correctness condition of the algorithm, which can be checked by induction, is that the values of P_i, T_i, and ψ_i must be the same for each iteration i in different *correct* processors.

Before starting a new iteration i, a processor first checks whether all the processors in P_{i-1} are correct or whether the total number of remaining tasks is no more than m/n. If either condition holds, it exits the loop. Then, if all the processors in P_{i-1} are correct, it computes a balanced distribution of the remaining set of tasks so that, overall, all the tasks are done by the processors in P_{i-1}. Otherwise the total number of remaining tasks is no more than m/n, and in that case the processor does all the remaining tasks itself. Overall, in either case, this implies $O(m)$ work.

If none of the above conditions hold, a new iteration i starts. The processor first calls the subroutine *Do_Work_and_Verify*. In this subroutine each processor in P_{i-1} gets allocated some subset of the tasks in T_{i-1} that it must execute, and a subset of the processors in P_{i-1} that it must *supervise*, that is, whose tasks it will *verify*. More formally,

Definition 4.1. For an iteration i of an execution of algorithm *Majority*, we say that a processor $p \in P_{i-1}$ supervises a processor $q \in P_{i-1}$, if p is assigned to verify all the tasks from T_{i-1} that q was assigned to perform in iteration i.

If a processor detects in this subroutine that some supervised processor in that subset is not doing the tasks it was assigned, it includes it in a set of faulty processors, returned as set Φ. We denote by $\Phi_{i,p}$ the processors that processor p suspects to be faulty at the *end* of subroutine *Do_Work_and_Verify* of iteration i. Then the processor calls the subroutine *Checkpoint*, which uses a check-pointing algorithm to combine the sets of suspected processors from all the processors in P^i into a common consistent set Φ_i; this denotes the consistent set of faulty processors at the *end* of iteration i. Finally, knowing which processors have been identified as faulty in this iteration, it updates the values of P_{i-1}, T_{i-1}, and ψ_{i-1} and obtains P_i, T_i, and ψ_i, respectively. Note that since $\psi_0 = f < n/2$ initially, at any point it is satisfied that $\psi_i < |P_i|/2$.

We now detail more the subroutines *Do_Work_and_Verify* and *Checkpoint*. We begin the first one. The code of subroutine *Do_Work_and_Verify* is shown in Fig. 6.

In the subroutine, W is an ordered list of tasks. We denote by W_i the value of W after the *end* of routine *Allocate_Tasks* of iteration i. Hence, W_i is an ordered list of tasks, all of them in T_{i-1}. This is needed to ensure that it is known the order in which a given processor is supposed to perform the tasks in its list W_i. That also allows us to ensure that all processors supervising a processor r verify the zth task allocated to r at the same time (and hence all find it either done or undone). Note also that to ensure that all correct processors finish the call to *Do_Work_and_Verify* at the same time, they all must be allocated the same number of tasks to perform.

Similarly, S is a sequence of sets $S[1], \ldots, S[\lceil \frac{2\psi}{v} \rceil + 2]$, each with at most v processors. We denote by S_i the value of S after the *end* of the routine *Allocate_Processors* of iteration i. These sets must also satisfy (in order for the same task to be verified at the same time by all the processors that do so) that the same processor r is in the same set $S_i[k]$ in all the

```
Do_Work_and_Verify(p, P, T, ψ, Φ):
1  W ← Allocate_Tasks(p, P, T)
2  S ← Allocate_Processors(p, P, T, ψ)
3  Φ ← ∅
4  for z = 1 to |W| do
5      perform the zth task in W
6      for k = 1 to ⌈2ψ/v⌉ + 2 do
7          verify the zth task of each processor in set S[k]
8          Φ ← Φ ∪ {r : l is the zth task allocated to r ∈ S[k] and was not done}
9      end for
10 end for
```

Fig. 6. Subroutine *Do_Work_and_Verify*. Code for processor p.

processors that supervise r. Then, all the tasks of r will be verified at the same time in the kth iteration of the inner "for" loop.

Let us now look at the allocation of tasks. For iteration i, we impose that $\lceil m_{i-1}/n_{i-1} \rceil$ different tasks from T_{i-1} are allocated to each processor in P_{i-1} by subroutine *Allocate_Tasks*, and that the number of processors allocated to execute two different tasks in T_{i-1} differs in at most one. Other than these, there are no other restrictions. For instance, if we number the tasks in T_{i-1} from 1 to m_{i-1} and the processors in P_{i-1} from 1 to n_{i-1}, the qth processor could be allocated the tasks with numbers $((kn_{i-1} + q - 1) \bmod m_{i-1}) + 1$, for $k = 0, \ldots, \lceil m_{i-1}/n_{i-1} \rceil - 1$.

We look now at the allocation of processors done in subroutine *Allocate_Processors*, for iteration i. We require that at least $2\psi_{i-1} + 1$ processors supervise any other processor (to be able to use Lemma 4.7, stated later). A processor implicitly supervises itself. Then, any deterministic function that assigns at least $2\psi_{i-1}$ other processors to each processor in P_{i-1} so that each processor is supervised by at least other $2\psi_{i-1}$ processors is valid. We also need to choose the sets S_{i-1} appropriately, as described above. All these could be done as follows. First, define a cyclic order in P_{i-1} and allocate to each processor the $2\psi_{i-1}$ processors that follow it in that order. Then, group the processors in sets of size v using the cyclic order and starting from some distinguished processor (e.g., the one with smallest PID). Number these sets from 1 to $\lceil n_{i-1}/v \rceil$. Each processor then gets assigned the sets that contain processors it has to supervise. To enforce that the same set is verified simultaneously, set number k is verified in the $(k \bmod (\lceil 2\psi_{i-1}/v \rceil + 2)) + 1$st iteration of the inner loop. Since $2\psi_{i-1}$ adjacent processors can span at most $\lceil 2\psi_{i-1}/v \rceil + 2$ sets (out of which at least $\lceil 2\psi_{i-1}/v \rceil + 1$ have v processors each), there is a way to schedule the verification of all the sets.

We now consider subroutine *Checkpoint*. Its code is detailed in Fig. 7. We denote by C_i the value of C at the *end* of the assignment at line 1 of the code, of iteration i. The subroutine uses two communication rounds. At iteration i, first each processor p sends its set $\Phi_{i,p}$ (computed in the subroutine *Do_Work_and_Verify*) to the processors in set C_i. Set C_i contains the first $2\psi_{i-1} + 1$ processors in P_{i-1} with the smallest PID. An elementary, but important, invariant of the algorithm is that set C_i is the same in all correct processors.

```
Checkpoint(p, P, ψ, Φ):
1  C ← the first 2ψ + 1 processors in P with smallest PID
2  send set Φ to every processor in C
3  if p ∈ C then
4      attempt to receive set Φ_q from each processor q ∈ P
5      Φ ← {b : processor b is in at least ψ + 1 received sets F_q}
6      send Φ to every processor in P
7  else
8      idle for the rest of the step
9      attempt to receive set Φ_c from each processor c ∈ C
10     Φ ← {b : processor b is in at least ψ + 1 received sets Φ_c}
11 end if
```

Fig. 7. Subroutine *Checkpoint*. Code for processor p.

The processors in C_i attempt to receive all sets $\Phi_{i,p}$ from the processors in P_{i-1}. Note that a faulty processor b may not send its corresponding set $\Phi^{i,b}$ or send an erroneous set $\Phi_{i,b}$. That is allowed and no note is taken of it by the correct processors. Also, messages received from processors not in P_{i-1} are disregarded by the correct processors. Only those processors that are in at least $\psi_{i-1}+1$ received sets from processors in P_{i-1} are considered faulty by the processors in set C_i. Then, the processors c in C_i send their updated sets $\Phi_{i,c}$ to the processors in P_{i-1}. Each processor p in P_{i-1} updates its set $\Phi_{i,p}$ by considering as faulty only the processors that are in at least $\psi_{i-1}+1$ received sets from processors in C_i and obtains Φ_i. Since P_{i-1} contains at least $2\psi_{i-1}+1$ processors, we have the following claim.

Lemma 4.7. *For an iteration i of an execution of algorithm Majority, if each processor in P_{i-1} is supervised by at least $2\psi_{i-1}+1$ different processors in P_{i-1}, then after subroutine Checkpoint has been executed, the set Φ_i is the same for every correct processor in P_{i-1}, it only contains faulty processors, and all the tasks allocated to processors in $P^i \setminus \Phi_i$ have been performed.*

Proof. Assuming the correct processors do the supervision properly, if some correct processor p detects a faulty processor q, and includes q in $\Phi_{i,p}$ in the subroutine *Do_Work_and_Verify*, then all correct processors that supervise q also do so. Then, each correct processor in C_i receives at least $\psi_{i-1}+1$ sets $\Phi_{i,p}$ containing q, since in any set of $2\psi_{i-1}+1$ processors (including the set of processors that supervised q) at least $\psi_{i-1}+1$ processors are correct. This also implies that the processors in P_{i-1} will receive at least $\psi_{i-1}+1$ sets $\Phi_{i,c}$ containing q (even if the faulty processors b in C_i send erroneous sets $\Phi_{i,b}$). Hence processor q will be in the final set Φ_i of each correct processor. Note that if processor q is not faulty and the faulty processors b send erroneous sets $\Phi_{i,b}$ that include q, q will not be included in a set Φ_i of a correct processor since there will not be more than ψ_{i-1} sets $\Phi_{i,b}$ containing q. Since this is true for each processor $q \in P_{i-1}$, after the subroutine *Checkpoint* has been executed the set Φ_i is the same for every correct processor in P_{i-1}, and it only contains faulty processors. This implies that the processors in $P_{i-1} \setminus \Phi_i$ performed the

tasks allocated to them correctly (otherwise they would not be in $P_{i-1} \setminus \Phi_i$ but in Φ_i). This completes the proof of the lemma. □

The following lemma shows that algorithm *Majority* solves the Do-All problem efficiently when $f < n/2$. Here $\phi \leqslant f$ is the exact number of faulty processors in the execution of interest of the algorithm. This value can be much smaller, for a particular execution, than the upper bound f.

Lemma 4.8. *Algorithm Majority, can be used to solve the* Do-All *problem in the failure model $\mathcal{F}_f$ with known f, $\phi \leqslant f$ actual Byzantine failures, and v task verifications per processor per step, with work $S = \mathrm{O}\big(m + mf/v + n(1 + f/v) \cdot \min\{\phi + 1, \log n\}\big)$ and message complexity $M = \mathrm{O}\big(n(f + 1) \cdot \min\{\phi + 1, \log n\}\big)$.*

Proof. It can be shown by induction that after each iteration i of the while loop of the algorithm, each correct processor has the same values of T_i, P_i, and $\psi_i \leqslant f$ and that the tasks not in T_i have been executed. Specifically, based on Lemma 4.7, if the correct processors begin an iteration i with common values of P_{i-1}, T_{i-1} and ψ_{i-1}, it follows that the (remaining) correct processors conclude this iteration with common values of P_i, T_i and ψ_i. Of course, initially all processors have the same P_0, T_0 and ψ_0. If there is at least one correct processor, then each iteration has a set T_i of smaller size. This implies that the algorithm terminates with all tasks performed and at least one correct processor being aware of this.

The proof of the work bound uses several ideas from [8]. To start, we adapt their Theorem 4 as follows. This theorem says that, under the crash failure model, if in every stage of a synchronous algorithm α the work to be performed is evenly divided among the processors, then the total number of stages executed in algorithm α is bounded by $\mathrm{O}(\log n)$. The proof uses the fact that the work previously assigned to a correct processor is not redone. We can adapt this proof to our algorithm, since we fully divide the work in each iteration and only redo tasks of failed processors. Hence, at most $\mathrm{O}(\log n)$ iterations are required.

We are going to study separately those iterations i of the while loop in which $m_{i-1} \geqslant n_{i-1}$ from those in which $m_{i-1} < n_{i-1}$. Since we assume $n \leqslant m$, initially $m_0 \geqslant n_0$. Furthermore, it is easy to show that once (if ever) $m_{i-1} < n_{i-1}$, this holds until the end of the execution as follows. Since less than half the processors in P_{i-1} can fail, if $m_{i-1} < n_{i-1}/2$, clearly at the end of the iteration i $m_i < n_i$. Otherwise, if $n_{i-1} > m_{i-1} \geqslant n_{i-1}/2$, then any task is assigned to at most two processors, and at the end of the iteration m_i has been reduced to less than half.

Then, we can consider both kind of iterations separately. Let us first consider iterations i of the while loop where $m_{i-1} \geqslant n_{i-1}$. Note that there is no such iteration in which more than $\lceil m/n \rceil$ tasks are allocated to any processor. This is so because initially $\lceil m/n \rceil$ tasks are allocated, and the number of failures required to have more than $\lceil m/n \rceil$ tasks in any other iteration is more than $n/2$. Hence, a faulty processor can force at most $\lceil m/n \rceil$ tasks to be redone. Thus, we have that at most $m + \phi\lceil m/n \rceil < 2m + \phi = \mathrm{O}(m)$ work spent executing tasks in these iterations. Similarly, in the iterations i where $m_{i-1} < n_{i-1}$, one task is allocated to each processor. We have from above that the number of iterations is $\mathrm{O}(\log n)$,

and it can be trivially observed that there can be at most $\phi + 1$ iterations. Hence, at most $O(n \cdot \min\{\phi + 1, \log n\})$ work is spent executing tasks in this case. Hence, in both kinds of iterations the work incurred in executing tasks is $O(m + n \cdot \min\{\phi + 1, \log n\})$. Since for each task executed there is one call to the checkpoint subroutine (each such call takes constant time) and at most $\lceil \frac{2f}{v} \rceil + 2$ verifications, the work bound follows. Note that the work incurred after the exit conditions of the while loop are satisfied is $O(m)$ (see discussion on the exit conditions in the description of the algorithm).

For the message bound, we use a similar argument. There are $O(\min\{\phi + 1, \log n\})$ iterations, with one call to the checkpoint subroutine in each, and at most $2n(2f + 1)$ messages required in each checkpoint call. The message complexity bound follows. Note that no communication takes place after the exit conditions of the while loop are satisfied. □

It is worth observing that in this case, communication helps improve work complexity.

4.1.4. **Algorithm** *Complete*

By combining the two cases considered by algorithms *Minority* and *Majority* for different ranges of *f*, we obtain an algorithm that efficiently solves Do-All for the entire range of *f*. We refer to this algorithm as algorithm *Complete*.

The correctness and the efficiency of algorithm *Complete* follows directly from the correctness and efficiency of algorithms *Minority* and *Majority*.

Theorem 4.9. *Algorithm Complete solves the* Do-All *problem in the failure model* $\mathcal{F}_f$ *with f known,* $\phi \leqslant f$ *actual Byzantine failures, and* v *verifications per processor per step, with work* $S = O(m + mn/v + n \log n/\Lambda_{n,\phi})$ *and no communication when* $f = \Omega(n)$*, and with work* $S = O(m + mf/v + n(1 + f/v) \cdot \min\{\phi + 1, \log n\})$ *and message complexity* $M = O(n(f + 1) \cdot \min\{\phi + 1, \log n\})$ *otherwise.*

4.2. The maximum number of faulty processors is unknown

In this section we assume that all we know about the number of faulty processors is that $f < n$. Using Lemma 4.2 and Theorem 4.3 of Section 4.1.1. we obtain the following lower bound.

Lemma 4.10. *Any fault-free execution of an algorithm that solves the* Do-All *problem in the failure model* $\mathcal{F}_f$ *with f unknown and with task verification, requires* $\Omega(m/n + m/v)$ *steps and* $\Omega(m + mn/v)$ *work.*

Proof. Since all that is known about the number of failures is that $f < n$, any algorithm that works under these assumptions has to work for $f = n - 1$. Then, the result follows from Lemma 4.2 and Theorem 4.3. □

Note that the lower bound of Lemma 4.1 does not depend on the knowledge of ϕ or *f* and is hence applicable to this case as well. Then, we have the following theorem.

Theorem 4.11. *Any algorithm that solves the* Do-All *problem in the failure model $\mathcal{F}_f$ with f unknown, in the presence of $\phi \leqslant f$ Byzantine failures, and with task verification, has work* $\Omega(m + mn/v + n \log n/\Lambda_{n,\phi})$.

Since *f* is unknown, a given algorithm must solve Do-All efficiently even for the case $f = n - 1$. Hence, if we use algorithm *Minority* assuming that $f = n - 1$, then Lemma 4.6 gives us an asymptotically matching upper bound on work for the setting that *f* is unknown. Taken together with the above lower bound result (Theorem 4.11), we conclude the following.

Corollary 4.12. *The work complexity of algorithm Minority in the failure model $\mathcal{F}_f$ with f unknown, $\phi \leqslant f$ actual Byzantine failures, and with task verification, is* $\Theta(m + mn/v + n \log n/\Lambda_{n,\phi})$.

Remark 4.2. In the conference version of this paper [11], the bound on the work for Minority for this setting was imprecisely given as $\Theta(mn/v)$, for any $n \leqslant m$ and $v \leqslant m$. As it can be observed from Corollary 4.12, this bound is valid only as long as $v = O(n)$ and $m/v = \Omega(\log n/\Lambda_{n,\phi})$.

5. Conclusions

In this paper we initiated the study of the Do-All problem under Byzantine processor failures. In particular we showed upper and lower bound results for synchronous message-passing processors prone to Byzantine failures for several cases. We considered the case where the maximum number of faulty processors *f* is known a priori, the case where *f* is not known, the case where tasks executions can be verified, and the case where task executions cannot be verified. We observed that in some cases work $\Theta(mn)$ (m number of tasks, n number of processors) is unavoidable. We also observed that in some cases communication does not help obtaining better work efficiency. In most cases we showed asymptotically matching upper and lower bound results. For the case where $f = o(n)$ and known, and task execution is verifiable, the upper bound, produced by the analysis of algorithm *Majority* is not tight. Obtaining tight bounds for this case is an interesting open question.

References

[1] C. Aguirre, J. Martinez-Munoz, F. Corbacho, R. Huerta, Small-world topology for multi-agent collaboration, in: Proc. 11th Internat. Workshop on Database and Expert Systems Appl., 2000, pp. 231–235.
[2] R.J. Anderson, H. Woll, Algorithms for the certified Write-All problem, SIAM J. Comput. 26 (5) (1997) 1277–1283.
[3] J. Buss, P.C. Kanellakis, P. Ragde, A.A. Shvartsman, Parallel algorithms with processor failures and delays, J. Algorithms 20 (1) (1996) 45–86.
[4] B. Chlebus, R. De Prisco, A.A. Shvartsman, Performing tasks on restartable message-passing processors, Distributed Computing 14 (1) (2001) 49–64.
[5] B. Chlebus, S. Dobrev, D. Kowalski, G. Malewicz, A.A. Shvartsman, I. Vrto, Towards practical deterministic Write-All algorithms, in: Proc. 13th ACM Symp. on Parallel Algorithms and Architectures (SPAA 2001), 2001, pp. 271–280.

[6] B.S. Chlebus, L. Gasieniec, D.R. Kowalski, A.A. Shvartsman, Bounding work and communication in robust cooperative computation, in: Proc. 16th Internat. Symp. Distributed Computing (DISC 2002), 2002, pp. 295–310.
[7] P. Dasgupta, Z. Kedem, M. Rabin, Parallel processing on networks of workstation: a fault-tolerant high performance approach, in: Proc. 15th IEEE Internat. Conf. Distributed Computer Systems (ICDCS 1995), 1995, pp. 467–474.
[8] R. De Prisco, A. Mayer, M. Yung, Time-optimal message-efficient work performance in the presence of faults, in: Proc. 13th ACM Symp. Principles of Distributed Computing (PODC 1994), 1994, pp. 161–172.
[9] S. Dolev, R. Segala, A.A. Shvartsman, Dynamic load balancing with group communication, Theoretical Computer Science, to appear. A preliminary version appears in the Proc. Sixth Internat. Colloquium on Structural Information and Communication Complexity (SIROCCO 1999), 1999, pp. 111–125.
[10] C. Dwork, J. Halpern, O. Waarts, Performing work efficiently in the presence of faults, SIAM J. Computing 27(5) (1998) 1457–1491. A preliminary version appears in the Proc. 11th ACM Symp. Principles of Distributed Computing (PODC 1992), 1992, pp. 91–102.
[11] A. Fernández, Ch. Georgiou, The Do-All problem with Byzantine processor failures, in: Proc. 10th Internat. Colloquium on Structural Information and Communication Complexity (SIROCCO 2003), 2003, pp. 117–132.
[12] Z. Galil, A. Mayer, M. Yung, Resolving message complexity of byzantine agreement and beyond, in: Proc. 36th IEEE Symp. Foundations of Computer Science (FOCS 1995), 1995, pp. 724–733.
[13] Ch. Georgiou, A. Russell, A.A. Shvartsman, Work-competitive scheduling for cooperative computing with dynamic groups, in: Proc. 35th ACM Symp. Theory of Computing (STOC 2003), 2003, pp. 251–258.
[14] Ch. Georgiou, A. Russell, A.A. Shvartsman, The complexity of synchronous iterative Do-All with crashes, Distributed Computing 17 (1) (2004) 47–63.
[15] Ch. Georgiou, A.A. Shvartsman, Cooperative computing with fragmentable and mergeable groups, J. Discrete Algorithms 1 (2) (2003) 211–235.
[16] J.F. Groote, W.H. Hesselink, S. Mauw, R. Vermeulen, An algorithm for the asynchronous Write-All problem based on process collision, Distributed Computing 14 (2) (2001) 75–81.
[17] P.C. Kanellakis, A.A. Shvartsman, Efficient parallel algorithms can be made robust, Distributed Computing 5(4) (1992) 201–217. A preliminary version appears in the Proc. Eighth ACM Symp. on Principles of Distributed Computing (PODC 1989), 1989, pp. 211–222.
[18] P.C. Kanellakis, A.A. Shvartsman, Fault-Tolerant Parallel Computation, Kluwer Academic Publishers, 1997.
[19] Z.M. Kedem, K.V. Palem, A. Raghunathan, P. Spirakis, Combining tentative and definite executions for dependable parallel computing, in: Proc. 23rd ACM Symp. on Theory of Computing (STOC 1991), 1991, pp. 381–390.
[20] E. Korpela, D. Werthimer, D. Anderson, J. Cobb, M. Lebofsky, SETI@home: Massively distributed computing for SETI, Comput. Sci. Engng. 3 (1) (2001) 78–83.
[21] L. Lamport, R. Shostak, M. Pease, The Byzantine generals problem, ACM Trans. Programming Languages and Systems 4 (3) (1982) 382–401.
[22] G. Malewicz, A. Russell, A.A. Shvartsman, Distributed cooperation during the absence of communication, in: Proc. 14th Internat. Symp. Distributed Computing (DISC 2000), 2000, pp. 119–133.
[23] C. Martel, A. Park, R. Subramonian, Work-optimal asynchronous algorithms for shared memory parallel computers, SIAM J. Comput. 21 (6) (1992) 1070–1099.
[24] C. Martel, R. Subramonian, On the complexity of certified Write-All algorithms, J. Algorithms 16 (3) (1994) 361–387.
[25] R.D. Schlichting, F.B. Schneider, Fail-stop processors: an approach to designing fault-tolerant computing systems, ACM Trans. Comput. Systems 1 (3) (1983) 222–238.
[26] M. Tambe, J. Adibi, Y. Alonaizon, A. Erdem, G.A. Kaminka, S. Marsella, I. Muslea, Building agent teams using an explicit teamwork model and learning, Artificial Intelligence 110 (2) (1999) 215–239.