The Complexity of Synchronous Iterative Do-All with Crashes*

Chryssis Georgiou[1], Alexander Russell[1], and Alex A. Shvartsman[1,2]

[1] Department of Computer Science and Engineering
University of Connecticut, Storrs, CT 06269, USA.
[2] Laboratory of Computer Science
Massachusetts Institute of Technology, Cambridge, MA 02139, USA.

Abstract. *Do-All* is the problem of performing N tasks in a distributed system of P failure-prone processors [8]. Many distributed and parallel algorithms have been developed for this problem and several algorithm simulations have been developed by iterating *Do-All* algorithms. The efficiency of the solutions for *Do-All* is measured in terms of *work* complexity where all processing steps taken by the processors are counted. We present the first non-trivial lower bounds for *Do-All* that capture the dependence of work on N, P **and** f, *the number of processor crashes*. For the model of computation where processors are able to make perfect load-balancing decisions locally, we also present matching upper bounds. We define the *r-iterative Do-All* problem that abstracts the repeated use of *Do-All* such as found in algorithm simulations. Our f-sensitive analysis enables us to derive a tight bound for *r-iterative Do-All* work (that is stronger than the r-fold work complexity of a single *Do-All*). Our approach that models perfect load-balancing allows for the analysis of specific algorithms to be divided into two parts: (i) the analysis of the cost of tolerating failures while performing work, and (ii) the analysis of the cost of implementing load-balancing. We demonstrate the utility and generality of this approach by improving the analysis of two known efficient algorithms. Finally we present a new upper bound on simulations of synchronous shared-memory algorithms on crash-prone processors.

1 Introduction

Performing a set of tasks in a decentralized setting is a fundamental problem in distributed computing. This is often challenging because the set of processors available to the computation and their ability to communicate may dynamically change due to perturbations in the computation medium. An abstract statement of this problem, referred to as the *Do-All* problem — P *fault-prone processors perform* N *independent tasks* — is one of the standard problems in the research on the complexity of fault-tolerant distributed computation [8, 17]. This

* This research is supported by the NSF Grant 9988304. The work of the second author is supported in part by a NSF Career Award. The work of the third author is supported in part by the NSF Career Award 9984774.

problem has been studied in shared-memory models (*Write-All*) [18, 19, 25], in message-passing models [6, 8, 10], and in partitionable networks (*Omni-Do*) [7, 11, 24]. 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 the design of practical distributed/parallel programs one needs to ensure good performance and dependability under unpredictable load patterns caused by deviations from synchrony or by the failures of some processors to complete tasks on time. Here again, a common challenge is to perform N independent tasks on P processors [16]. Such tasks could be copying a large array, searching a collection of data, or applying a function to all elements of a matrix [14].

In this paper we focus on the work complexity of the *Do-All* problem in the presence of arbitrary failure patterns imposed by an adversary. The processors are synchronous and are assumed to be fail-stop [30]. The work complexity reflects the *total* amount of processing steps expended by an algorithm [18]. A distinguishing feature of our results is that the complexity is expressed in terms of the *number of processor crashes* f in addition to P and N.

Our approach is motivated in part by the analyses of the consensus problems. The venerable FLP impossibility result [9] and the algorithms that solve consensus in the models that allow fault-tolerant solutions teach the following: (i) asynchronous models are too weak for fault-tolerance [22], and (ii) the maximum number of processor failures needs to be included in upper/lower bounds and impossibility results, e.g., to tolerate f failures in some models, consensus algorithms require $P = 3f + 1$ total processors [23, 28]. In this work we consider crash failures to ensure that solutions exist for as long as the number of failures f is inferior to the number of processors P, and we aim to express the work of the synchronous processors as a function of N, P and f.

Until very recently, an unsatisfactory landscape existed with respect to the understanding of how the bounds on work depend on f, the number of failures. That is, work was typically given as a function of N and P, but it was either not elucidated how f impacts work, or, when f was a part of the equation, it was primarily due to the nature of a specific algorithm, and not due to the inherent properties of the *Do-All* problem. For example, the work of the best known synchronous shared-memory algorithm [17] is given as a function of N and P. This is also the case with the best known asynchronous shared-memory algorithm [2]. Similarly, the best known shared-memory lower bound on work for *Do-All* is not parameterized in terms of f [19]. Likewise, the best known lower bound applicable to message-passing models does not involve f [3]. The work of message-passing algorithms, e.g. [6, 10], typically includes f, but this is due to the use of single coordinators, which means that for f coordinator failures the work necessarily includes a factor $f \cdot P$. A message-passing algorithm using multiple coordinators [4] avoids this inefficiency and includes a factor that depends on $\log f$ (but as we show in this paper, that analysis involves f in a somewhat superficial way). Thus prior lower/upper bound results for *Do-All* do not teach adequately how the work complexity depends on the number of failures f.

When considering the synchronous shared-memory computing with failure-prone processors the impact of imprecise analysis of work complexity is especially significant. Approaches such as [21, 29] use *iterative Do-All* approach to execute synchronous parallel (PRAM) algorithms on failure-prone processors by simulating the parallel steps of ideal processors with the help of some chosen *Do-All* algorithm (see also related work below). It was shown that the execution of a single N-processor step on P failure-prone processors does not exceed the complexity of solving a N-size instance of *Do-All* using P failure-prone processors. Thus if $W_{N,P}$ is the complexity of solving a *Do-All* instance of size N using P processors, and the *parallel-time*$\times$*processor* product of the given N-processor algorithm is $\tau \cdot N$, then the algorithm can be deterministically simulated with work $O(\tau \cdot W_{N,P})$. If the analysis does not accurately reflect the impact of the number of failures f, then the resulting upper bound is needlessly inflated.

Contributions. In this work we study the work complexity of deterministic *Do-All* in the presence of arbitrary dynamic patterns of stop-failures. Let N be the size of the *Do-All* problem, P the number of processors, and f the number of crashes ($0 \le f < P \le N$). We present the first *complete* analysis of *Do-All* work complexity under the perfect load balancing assumption by proving matching upper and lower bounds as functions of N, P *and* f. This is for the model of computation where the computation is fully abstracted away from the low-level shared-memory and message-passing issues, and where a worst-case omniscient dynamic adversary can cause up to f crashes. This also establishes the first non-trivial lower bound for *Do-All* for moderate number of failures ($f \le P/\log P$). An important contribution of this work is the definition and analysis of the *r-iterative Do-All* problem that models the repetitive use of *Do-All* algorithms (such as found in algorithm simulations).

We demonstrate the utility and generality of our results by showing new bounds on work for fault-tolerant simulations of arbitrary PRAM algorithms on crash-prone processors, and by improving the analyses of two known algorithms. We derive a new and complete failure sensitivity analysis of the best known algorithm for the synchronous shared-memory model (algorithm W [17]). We also give an improved analysis of the work and message complexity for an efficient message-passing algorithm (algorithm AN [4]).

We give a detailed summary of the complexity results in Section 2.

Related work—algorithm simulations. *Do-All* algorithms can be used iteratively to simulate parallel algorithms formulated for synchronous failure-free processors in deterministic and probabilistic settings [21, 19, 26, 27, 29]. This commonly requires that (i) the individual processor steps are made idempotent (since they may have to be performed multiple times due to failures or asynchrony), and that (ii) a linear in the number of processors auxiliary memory is made available (to be used as a "scratchpad" and to store intermediate results). While the former can be solved with the help of an automated tool, e.g., a compiler, the latter requires sophisticated solutions because of the difficulty of (re)using the auxiliary memory due to "late writers" (i.e., processors that are slow and that unknowingly write stale values to memory). Examples of randomized solutions addressing these problems include [1, 20]. Another important aspect of algorithm simulations is the use of on optimistic approach, where the computation may proceed for several steps assuming that all tasks assigned to active processors are successfully completed, e.g., [19]. In some deterministic models optimal simulations are possible (cf. [29]), however randomized solutions are able to achieve optimality (whp) for broader ranges of models and algorithms. An example of a practical implementation is discussed in [5].

The rest of the paper is structured as follows. In Section 2 we summarize the results. Section 3 we give models and definitions. In Section 4 we present the bounds under the perfect load-balancing assumption. We give new upper bounds for the message-passing model in Section 5, and for the shared-memory model in Section 6. We conclude in Section 7.

Due to the page limit, most proofs are either omitted or given as sketches. A full version of this paper is available [13].

2 Grand Tour of the Results

We let *Do-All*(N, P, f) stand for the *Do-All* problem for N tasks, P processors and up to f failures. We let *Do-All*$^{\mathcal{O}}(N, P, f)$ denote the *Do-All*(N, P, f) problem that is solved with the use of an omniscient oracle that assists the processors (but unlike the oracle's delphian colleague, it cannot predict the future). The oracle assumption is used as a *tool* for studying the work complexity patterns of *any* fault-tolerant algorithm that implements perfect work-load balancing. This allows for the complexity analysis of specific algorithms to be divided into two parts: (i) the analysis of the cost of tolerating failures while performing work assuming perfect load-balancing, and (ii) the analysis of the cost of implementing perfect load-balancing. We use exactly this approach to derive new f-sensitive upper bounds for message-passing and shared-memory models.

We have shown [12, 18] that *Do-All*$^{\mathcal{O}}(N, P, f)$ can be solved with work $O(N + P\frac{\log P}{\log\log P})$ where $f < P$, and gave a matching lower bound in the specific case where $f = \frac{P}{\log\log P} + O(\frac{P}{(\log\log P)^2})$. This meant that as long as the adversary can cause at least $\frac{P}{\log\log P}$ failures, *Do-All*$^{\mathcal{O}}(N, P, f)$ has matching upper and lower bounds of $N + P\frac{\log P}{\log\log P}$. We also showed that when $f = o(\frac{P}{\log P})$ then *Do-All*$^{\mathcal{O}}(N, P, f)$ can be solved with work $O(N + P\log_{\max\{2, \frac{P}{2f}\}} P)$.

Thus prior to our newest results: (i) no non-trivial lower bounds were known for $f < \frac{P}{\log P}$, (ii) no f-sensitive analysis was available for the upper bounds when f is between $\frac{P}{\log P}$ and $\frac{P}{\log\log P}$, and therefore, (iii) there existed a gap in upper/lower bounds analysis for the range $1 < f < \frac{P}{\log\log P}$, where $f = \omega(1)$. Yet practical concerns would be well served by the knowledge of what happens in *Do-All* when the number of failures is moderate. In particular, it is important to understand the behavior of the best algorithms for the entire range of f.

The detailed contributions in this work are as follows.

I. We provide upper bounds (Section 4.1) and matching lower bounds (Section 4.2) that address *all* remaining gaps, hence we give a *complete* analysis of *Do-All*$^{\mathcal{O}}(N, P, f)$ for the *entire* range of f. The bounds on work W are:[1]

$$\begin{aligned} (a)\ W &= \{O|\Omega\}\left(N + P\frac{\log P}{\log\log P}\right) \ \text{ when } f > c\frac{P}{\log P},\ \text{any } c > 0, \\ (b)\ W &= \{O|\Omega\}\left(N + P\log_{\frac{P}{f}} P\right) \ \text{ when } f \le c\frac{P}{\log P},\ \text{any } c > 0. \end{aligned} \tag{1}$$

The lower bounds of course apply to algorithms in weaker models.

The quantity $\mathcal{Q}_{P,f}$, defined below and extracted from the bounds (1)(a,b) above, plays an important role in the analysis of complexity of several algorithms.

$$\mathcal{Q}_{P,f} = \begin{cases} P\frac{\log P}{\log\log P} \text{ when } f > c\frac{P}{\log P},\ \text{any } c > 0, \\ P\log_{\frac{P}{f}} P \text{ when } f \le c\frac{P}{\log P},\ \text{any } c > 0. \end{cases} \tag{2}$$

We use our bounds (1) to derive new bounds for algorithms where the extant analyses do not integrate f adequately. This is done by analyzing how the work-load balancing is implemented by the algorithms, e.g., by using coordinators or global data-structures. We show the following.

II. In Section 5.1 we provide new analysis of algorithm AN of Chlebus *et al.* [4] for *Do-All* in the message-passing model with crashes. This algorithm has best known work for moderate number of failures. We show the complete analysis of work W and message complexity M:

$$W = O\left(\log f\left(N + \mathcal{Q}_{P,f}\right)\right) \text{ and } M = O\left(N + \mathcal{Q}_{P,f} + fP\right).$$

III. In Section 6.1 we give a *complete* analysis of the work complexity W of the algorithm of Kanellakis and Shvartsman [17] that solves the *Do-All* (*Write-All*) problem in synchronous shared-memory systems with processor crashes:

$$W = O\left(N + \mathcal{Q}_{P,f}\log N\right).$$

Note that the two algorithms [4, 17] are designed for different models and use dissimilar data and control structures, however both algorithms make their load-balancing decisions by gathering global knowledge. By understanding what work is expended on load balancing vs. the inherent work overhead due to the lower bounds (1), we are able to obtain the new results while demonstrating the utility and the generality of our approach.

Do-All algorithms have been used in developing *simulations* of failure-free algorithms on failure-prone processors, e.g., [21, 29]. This is done by iteratively using a *Do-All* algorithm to simulate the steps of the failure-free processors. In this paper we abstract this idea as the *iterative Do-All* problem as follows:

[1] We let "$\{O|\Omega\}$" stand for "O" when describing an upper bound and for "Ω" when describing a lower bound. All logarithms are to the base 2 unless explicitly specified otherwise. The expression $\log X$ stands for $\max\{1,\ \log_2 X\}$ for the given X in the description of complexity results.

The *r-iterative Do-All*(N,P,f), or *r-Do-All*(N,P,f), is the problem of using P processors to solve r instances of N-task *Do-All* by doing one set of tasks at a time.

The oracle *r-Do-All*$^{\mathcal{O}}(N,P,f)$ is defined similarly. An obvious solution for this problem is to run a *Do-All* algorithm r times. If the work complexity of *Do-All* in a given model is $W_{N,P,f}$, then the work of *r-Do-All* is clearly no more than $r \cdot W_{N,P,f}$. We present a substantially better analysis:

IV. In Section 4.3 we show matching upper and lower bounds on work W for *r-Do-All*$^{\mathcal{O}}(N,P,f)$, where $f < P \le N$, for specific ranges of failures.

$$\begin{aligned} &(a)\ W = \{O|\Omega\}\left(r \cdot \left(N + P\frac{\log P}{\log\log P}\right)\right) \text{ when } f > \frac{Pr}{\log P}, \\ &(b)\ W = \{O|\Omega\}\left(r \cdot \left(N + P\frac{\log P}{\log \frac{Pr}{f}}\right)\right) \text{ when } f \le \frac{Pr}{\log P}. \end{aligned} \qquad (3)$$

We extract the quantity $\mathcal{R}_{r,P,f}$, defined below, from the bounds (3)(a,b) above, as it plays an important role in the analysis of iterative *Do-All* algorithms.

$$\mathcal{R}_{r,P,f} = \begin{cases} P\frac{\log P}{\log\log P} \text{ when } f > \frac{Pr}{\log P}, & (a) \\ P\frac{\log P}{\log \frac{Pr}{f}} \text{ when } f \le \frac{Pr}{\log P}. & (b) \end{cases} \qquad (4)$$

Note that for any r we have $\mathcal{Q}_{P,f} \ge \mathcal{R}_{r,P,f}$, and for the specific range of f in (4)(b) we have that $\mathcal{Q}_{P,f} = \omega(\mathcal{R}_{r,P,f})$ with respect to r (fixed P and f). Thus our bounds (3) are asymptotically better than those obtained by computing the product of r and the (non-iterated) *Do-All* bounds (1).

V. In Section 5.2 we show how to solve *r-Do-All*(N,P,f) on synchronous message-passing processors with the following work (W) and message (M) complexity.

$$W = O\left(r \cdot \log\tfrac{f}{r} \cdot (N + \mathcal{R}_{r,P,f})\right) \text{ and } M = O\left(r \cdot (N + \mathcal{R}_{r,P,f}) + fP\right).$$

VI. In Section 6.2 we use *r-Do-All*(N,P,f) to show that P processors with crashes can simulate any synchronous N-processor, r-time shared-memory algorithm (PRAM) with work:

$$W = O\left(r \cdot (N + \mathcal{R}_{r,P,f} \log N)\right).$$

This last result is strictly better than the previous deterministic bounds for parallel algorithm simulations using the *Do-All* algorithm [17] (the best known to date) and simulation techniques such as [21, 29] (due of the the relationship between $\mathcal{Q}_{P,f}$ and $\mathcal{R}_{r,P,f}$ as pointed out above).

3 Models and Definitions

We define the models, the abstract problem of performing N tasks in a distributed environment consisting of P processors that are subject to stop-failures, and the work complexity measure.

Distributed setting. We consider a distributed system consisting of P synchronous processors. We assume that P is fixed and is known. Each processor has a unique identifier (PID) and the set of PIDs is totally ordered.

Tasks. We define a *task* to be a computation that can be performed by any processor in at most one time step; its execution does not dependent on any other task. The tasks are also *idempotent*, i.e., executing a task many times and/or concurrently has the same effect as executing the task once. Tasks are uniquely identified by their task identifiers (TIDs) and the set of TIDs is totally ordered. We denote by $\mathcal{T}$ the set of N tasks and we assume that $\mathcal{T}$ is known to all the processors.

Model of failures. We assume the *fail-stop* processor model [30]. A processor may crash at any moment during the computation and once crashed it does not restart. We let an omniscient *adversary* impose failures on the system, and we use the term *failure pattern* to denote the set of the events, i.e., crashes, caused by the adversary. A *failure model* is then the set of all failure patterns for a given adversary. For a failure pattern F, we define the *size* f of the failure pattern as $f = |F|$ (the number of failures).

The Oracle model. In Section 4 we consider computation where processors are assisted by a deterministic omniscient *oracle*. Any processor may contact the oracle once per step. The introduction of the oracle serves two purposes.
(1) The oracle strengthens the model by providing the processors with any information about the progress of the computation (the oracle cannot predict the future). Thus the lower bounds established for the oracle model also apply to any weaker model, e.g., without an oracle.
(2) The oracle abstracts away any concerns about communication that normally dominate specific message-passing and shared-memory models. This allows for the most general results to be established and it enables us to use these results in the context of specific models by understanding how the information provided by an oracle is simulated in specific algorithms.

Communication. In Sections 5 and 6 we deal with message-passing and shared-memory models. For computation in the message-passing model, we assume that there is a known upper bound on message delays. (Communication complexity is defined in Section 5.) When considering computation in the shared-memory model, we assume that reading or writing to a memory cell takes one time unit, and that reads and writes can be concurrent.

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

> *Do-All: Given a set $\mathcal{T}$ of N tasks and P processors, perform all tasks for any failure pattern in the failure model $\mathcal{F}$.*

We let *Do-All*(N, P, f) stand for the *Do-All* problem for N tasks, P processors ($P \leq N$), and any pattern of crashes F such that $|F| \leq f < P$. We let *Do-All*$^{\mathcal{O}}(N, P, f)$ stand for the *Do-All*(N, P, f) problem with the oracle. We define the *iterative* *Do-All* problem as follows:

Iterative Do-All: Given any r sets $\mathcal{T}_1, \ldots, \mathcal{T}_r$ of N tasks each and P processors, perform all $r \cdot N$ tasks, doing one set at a time, for any failure pattern in the failure model $\mathcal{F}$.

We denote such *r-iterative Do-All* by *r-Do-All*(N, P, f). The oracle version *r-Do-All*$^{\mathcal{O}}(N, P, f)$ is defined similarly.

Measuring efficiency. We are interested in studying the complexity of *Do-All* measured as *work* (cf. [17, 8, 6]). We assume that it takes a unit of time for a processor to perform a unit of work, and that a single task corresponds to a unit of work. Our definition of *work complexity* is based on the *available processor steps* measure [18]. Let $\mathcal{F}$ be the adversary model. For a computation subject to a failure pattern F, $F \in \mathcal{F}$, denote by $P_i(F)$ the number of processors completing a unit of work in step i of the computation.

Definition 1. *Given a problem of size N and a P-processor algorithm that solves the problem in the failure model $\mathcal{F}$, if the algorithm solves the problem for a pattern F in $\mathcal{F}$, with $|F| \leq f$, by time step τ, then the work complexity W of the algorithm is:* $W_{N,P,f} = \max_{F \in \mathcal{F},\ |F| \leq f} \left\{ \sum_{i \leq \tau} P_i(F) \right\}$.

Note that the idling processors consume a unit of work per step even though they do not contribute to the computation. Definition 1 does not depend on the specifics of the target model of computation, e.g., whether it is message-passing or shared-memory. (Communication complexity is defined similarly in Section 5.)

4 The Bounds with Perfect Load Balancing

In this section we give the complete analysis of the upper and lower bounds for the *Do-All*$^{\mathcal{O}}(N, P, f)$ and *r-Do-All*$^{\mathcal{O}}(N, P, f)$ problems for the entire range of f crashes ($f < P \leq N$). (Note: we use the quantities $\mathcal{Q}_{P,f}$ and $\mathcal{R}_{r,P,f}$ that are defined in Section 2 in equations (2) and (4) respectively.)

4.1 Do-All Upper Bounds

To study the upper bounds for *Do-All* we give an oracle-based algorithm in Figure 1. The oracle tells each processor whether or not all tasks have been performed *Oracle-says*(), and what task to perform next *Oracle-task*() (the correctness of the algorithm is trivial). Thus the oracle performs the termination and load-balancing computation on behalf of the processors.

```
for each processor PID = 1..P begin
    global T[1..N];
    while Oracle-says(PID) = "not done"
        do perform task T[Oracle-task(PID)] od
end.
```

Fig. 1. Oracle-based algorithm.

Lemma 1. $[18, 12]$ *The Do-All*$^{\mathcal{O}}(N,P,f)$ *problem with* $f < P \leq N$ *can be solved using work* $W = O\left(N + P\frac{\log P}{\log\log P}\right)$.

Note that Lemma 1 does not teach how work depends on the number of crashes f.

Lemma 2. *For any* $c > 0$, *Do-All*$^{\mathcal{O}}(N,P,f)$ *can be solved for any stop-failure pattern with* $f \leq c\frac{P}{\log P}$ *using work* $W = O(N + P + P\log_{P/f} N)$.

Proof. The proof is based on the proof of Theorem 3.6 of [12].

We now give our main upper-bound result.

Theorem 1. *Do-All*$^{\mathcal{O}}(N,P,f)$ *can be solved for any failure pattern using work* $W = O(N + \mathcal{Q}_{P,f})$.

Proof. This follows directly from Lemmas 1 and 2.

4.2 Do-All Lower Bounds

We now show matching lower bounds for *Do-All*$^{\mathcal{O}}(N,P,f)$. Note that the results in this section hold also for the *Do-All*(N,P,f) problem (without the oracle).

Lemma 3. $[17, 12]$ *For any algorithm that solves Do-All*$^{\mathcal{O}}(N,P,f)$ *there exists a pattern of* f *stop-failures* ($f < P$) *that results in work* $W = \Omega\left(N + P\frac{\log P}{\log\log P}\right)$.

We now define a specific adversarial strategy used to derive our lower bounds. Let Alg be any algorithm that solves the *Do-All* problem. Let P_i be the number of processors remaining at the end of the i^{th} step of Alg and let U_i denote the number of tasks that remain to be done at the end of step i. Initially, $P_0 = P$ and $N = U_0$. Define $\kappa = \frac{f\log(\frac{P}{f})}{P\log P}$, $0 < \kappa < 1$.

Adversary Adv: At step i ($i \geq 1$) of Alg, the adversary stops processors as follows: Among U_{i-1} tasks remaining after the step $i-1$, the adversary chooses $U_i = \lfloor \kappa U_{i-1} \rfloor$ tasks with the least number of processors assigned to them and crashes these processors. The adversary continues for as long as $U_i > 1$. As soon as $U_i = 1$ the adversary allows all remaining processors to perform the single remaining task, and Alg terminates.

Lemma 4. *Given any* $c > 0$ *and any algorithm* Alg *that solves Do-All*$^{\mathcal{O}}(N,P,f)$ *for* $P \geq N$, *the adversary* Adv *causes* f *stop-failures,* $f \leq c\frac{P}{\log P}$, *and* $W = \Omega(P + P\log_{\frac{P}{f}} N)$.

Proof. (Sketch.) Let τ be the number of iterations caused by adversary Adv and P_τ be the number of processors at the last iteration of Alg. Given the definition of Adv (given above), we find that $\tau \geq \frac{\log N}{\log\frac{P}{f} + \log\log f}$ and $P_\tau > P - f$. The result follows by the observation that the work caused by Adv is at least $\tau \cdot P_\tau$.

Lemma 5. *Given any $c > 0$ and any algorithm Alg that solves Do-All$^{\mathcal{O}}(N, P, f)$ for $P \le N$, there exists an adversary that causes f stop-failures, $f \le c\frac{P}{\log P}$, and $W = \Omega(N + P \log_{\frac{P}{f}} P)$.*

We now give our main lower-bound result.

Theorem 2. *Given any algorithm Alg that solves Do-All$^{\mathcal{O}}(N, P, f)$ for $P \le N$, there exists an adversary that causes $W = \Omega(N + \mathcal{Q}_{P,f})$.*

Proof. For the range of failures $0 < f \le c\frac{P}{\log P}$ Lemma 5 establishes the bound. When $f = c\frac{P}{\log P}$, the work (per Lemma 5) is $\Omega\left(N + P\frac{\log P}{\log\log P}\right)$. For larger f the adversary establishes this worst case work using the initial $c\frac{P}{\log P}$ failures.

4.3 Iterative Do-All

Do-All algorithms have been used in developing simulations of failure-free algorithms on failure-prone processors. This is done by iteratively using a *Do-All* algorithm to simulate the steps of the failure-free processors. We study the *iterative Do-All* problems to understand the complexity implications of iterative use of *Do-All* algorithms.

In principle *r-Do-All*(N, P, f) can be solved by running an algorithm for *Do-All*(N, P, f) r times. If the work of a *Do-All* solution is W, then the work of the *r-iterative Do-All* is at most $r \cdot W$. However we show that it is possible to obtain a finer result. We refer to each *Do-All* iteration as the *round* of *r-Do-All*$^{\mathcal{O}}(N, P, f)$.

Theorem 3. *r-Do-All$^{\mathcal{O}}(N, P, f)$ can be solved with $W = O(r \cdot (N + \mathcal{R}_{r,P,f}))$.*

Proof. (Sketch) In the case where $f > \frac{Pr}{\log P}$, we can have $\Theta(\frac{P}{\log P})$ processor failures in all r rounds. From this and Theorem 1, the work is $O(r \cdot (N + P\frac{\log P}{\log\log P}))$. Consider the case where $f \le \frac{Pr}{\log P}$. Let f_i be the number of failures in round r_i. It is easily shown that the work in every round is $O(N + P\frac{\log P}{\log \frac{P}{f_i}})$. We treat f_i as a continuous parameter and we compute the first derivative and second derivative w.r.t. f_i. The first derivative is positive, the second derivative is negative. Hence the first derivative is decreasing (with f_i). In this case, given any two f_i, f_j where $f_i > f_j$, the failure pattern obtained by replacing f_i with $f_i - \epsilon$ and f_j by $f_j + \epsilon$ (where $\epsilon < (f_i - f_j)/2$) results in increased work. This implies that the work is maximized when all f_is are equal, specifically when $f_i = f/r$. This results to total work of $O(r \cdot (N + P\frac{\log P}{\log \frac{Pr}{f}}))$.

Theorem 4. *Given any algorithm that solves r-Do-All$^{\mathcal{O}}(N, P, f)$ there exists a stop-failure adversary that causes $W = \Omega(r \cdot (N + \mathcal{R}_{r,P,f}))$.*

Proof. In the case where $f > \frac{Pr}{\log P}$ the adversary may fail-stop $\frac{P}{\log P}$ processors in every round of *r-Do-All*$^{\mathcal{O}}(N, P, f)$. Note that for this adversary $\Omega(P)$ processors remain alive during the first $\lceil r/2 \rceil$ rounds. Per Theorem 2 this results in $\lceil r/2 \rceil \cdot \Omega(N + P\frac{\log P}{\log\log P}) = \Omega(Nr + Pr\frac{\log P}{\log\log P})$ work. In the case where $f \le \frac{Pr}{\log P}$

the adversary ideally would kill $\frac{f}{r}$ processors in every round. It can do that in the case where f divides r. If this is not the case, then the adversary kills $\lceil \frac{f}{r} \rceil$ processors in r_A rounds and $\lfloor \frac{f}{r} \rfloor$ in r_B rounds in such a way that $r = r_A + r_B$. Again considering the first half of the rounds and appealing to Theorem 2 results in a $\Omega(r(N + P \log_{\frac{rP}{f}} P))$ lower bound for work. Note that we consider only the case where $r \leq f$; otherwise the work is trivially $\Omega(rN)$.

5 New Bounds for the Message-Passing Model

In this section we demonstrate the utility of the complexity results under the perfect load-balancing assumption by giving a tight and complete analysis of the algorithm AN [4] and establish new complexity results for the iterative *Do-All* in the message-passing model.

The efficiency of message-passing algorithms is characterized in terms of their work and message complexity. We define message complexity similarly to Definition 1 of work: For a computation subject to a failure pattern F, $F \in \mathcal{F}$, denote by $M_i(F)$ the number of point-to-point messages sent during step i of the computation. For a given problem of size N, if the computation solves the problem by step τ in the presence of the failure pattern F, where $|F| \leq F$, then the message complexity M is $M_{N,P,f} = \max_{F\in\mathcal{F},\ |F|\leq f} \left\{ \sum_{i\leq\tau} M_i(F) \right\}$.

5.1 Analysis of Algorithm AN

Algorithm AN presented by Chlebus *et al.* [4] uses a multiple-coordinator approach to solve *Do-All*(N, P, f) on crash-prone synchronous message-passing processors ($P \leq N$). The model assumes that messages incur a known bounded delay and that reliable multicast [15] is available, however messages to/from faulty processors may be lost.

Description of algorithm AN. Due to the space limitation, we give a very brief description of the algorithm; for additional details we refer the reader to [4]. Algorithm AN proceeds in a *loop* which is iterated until all the tasks are executed. A single iteration of the loop is called a *phase*. A phase consists of three consecutive *stages*. Each stage consists of three steps. In each stage processors use the first step to receive messages sent in the previous stage, the second step to perform local computation, and the third step to send messages. A processor can be a *coordinator* or a *worker*. A phase may have multiple coordinators. The number of processors that assume the coordinator role is determined by the *martingale principle*: if none of the expected coordinators survive through the entire phase, then the number of coordinators for the next phase is doubled. If at least one coordinator survives in a given phase, then in the next phase there is only one coordinator. A phase that is completed with at least one coordinator alive is called *attended*, otherwise it is called *unattended*.

Processors become coordinators and balance their loads according to each processor's *local view*. A local view contains the set of ids of the processors assumed to be alive. The local view is partitioned into *layers*. The first layer contains one processor id, the second two ids, the i^{th} contains 2^{i-1} ids.

Given a phase, in the first stage, the processors perform a task according to the load balancing rule derived from their local views and report the completion of the task to the coordinators of that phase (determined by their local views). In the second stage, the coordinators gather the reports, they update the knowledge of the done tasks and they multicast this information to the processors that are assumed to be alive. In the last stage, the processors receive the information sent by the coordinators and update their knowledge of done tasks and their local views. Given the full details of the algorithm, it is not difficult to see that the combination of coordinators and local views allows the processors to obtain the information that would be available from the oracle in the algorithm in Figure 1.

It is shown in [4] that the work of algorithm AN is $W = O((N + N \log N / \log\log N) \log f)$ and its message complexity is $M = O(N + P \log P / \log\log P + fP)$.

New analysis of work complexity. To assess the work W, we consider separately all the attended phases and all the unattended phases of the execution. Let W_a be the part of W spent during all the attended phases and W_u be the part of W spent during all the unattended phases. Hence we have $W = W_a + W_u$.

Lemma 6. [4] *In any execution of algorithm* AN *with* $f < P$ *we have* $W_a = O(N + P \frac{\log P}{\log\log P})$ *and* $W_u = O(W_a \log f)$.

We now give the new analysis of algorithm AN.

Lemma 7. *In any execution of algorithm* AN *we have* $W_a = O(N + P \log_{\frac{P}{f}} P)$, *when* $f \le c\frac{P}{\log P}$, *for any* $c > 0$.

Theorem 5. *In any execution of algorithm* AN *we have* $W = O(\log f (N + \mathcal{Q}_{P,f}))$.

Proof. This follows from Lemmas 6 and 7 and the fact that $W = W_a + W_u$.

Analysis of message complexity. To assess the message complexity M we consider separately all the attended phases and all the unattended phases of the execution. Let M_a be the number of messages sent during all the attended phases and M_u the number of messages sent during all the unattended phases. Hence we have $M = M_a + M_u$.

Lemma 8. [4] *In any execution of algorithm* AN *we have* $M_a = O(W_a)$ *and* $M_u = O(fP)$.

Theorem 6. *In any execution of algorithm* AN *we have* $M = O(N + \mathcal{Q}_{P,f} + Pf)$.

Proof. It follows from Lemmas 6, 7 and 8 and the fact that $M = M_a + M_u$.

5.2 Analysis of Message-Passing Iterative Do-All

We now consider the message-passing r-*Do-All*(N, P, f) problem ($P \le N$).

Theorem 7. *The* r-*Do-All*(N, P, f) *problem can be solved on synchronous crash-prone message-passing processors with work* $W = O(r \cdot \log \frac{f}{r} \cdot (N + \mathcal{R}_{r,P,f}))$ *and with message complexity* $M = O(r \cdot (N + \mathcal{R}_{r,P,f}) + fP)$.

6 New Bounds for the Shared-Memory Model

Here we give a new refined analysis of the most work-efficient known *Do-All* algorithm for the shared-memory model, algorithm W [17]. We also establish the complexity results for the iterative *Do-All* and for simulations of synchronous parallel algorithms on crash-prone processors.

6.1 Analysis of Algorithm W

Algorithm W solves *Do-All*(N, P, f) in the shared-memory model (where *Do-All* is better known as *Write-All*). Its work for any pattern of crashes is $O(N + P \log N \log P / \log\log P)$ for $P \leq N$ [17]. Note that this bound is conservative, since it does not include f, the number of crashes.

Description of the algorithm. We now give a brief description of the algorithm; for additional details we refer the reader to [18]. Algorithm W is structured as a parallel *loop* through four phases: (W1) a failure detecting phase, (W2) a load rescheduling phase, (W3) a work phase, and (W4) a phase that estimates the progress of the computation, the remaining work and that controls the parallel loop. These phases use full binary trees with $O(N)$ leaves. The processors traverse the binary trees top-down or bottom-up according to the phase. Each such traversal takes $O(\log N)$ time (the height of a tree). For a single processor, each iteration of the loop is called a *block-step*; since there are four phases with at most one tree traversal per phase, each block step takes $O(\log N)$ time.

In algorithm W the trees stored in shared memory serve as the gathering places for global information about the number of active processors, remaining tasks and load balancing. It is not difficult to see that these binary trees indeed provide the information to the processors that would be available from an oracle in the oracle model. The binary tree used in phase W2 to implement load balancing and phase W3 to assess the remaining work is called the *progress tree*.

Here we use the parameterized version of the algorithm with $P \leq N$ and where the progress tree has $U = \max\{P, N/\log N\}$ leaves. The tasks are associated with the leaves of this tree, with N/U tasks per leaf. Note that each block-step still takes time $O(\log N)$.

New complexity analysis. We now give the work analysis. We charge each processor for each block step it starts, regardless of whether or not the processor completes it or crashes.

Lemma 9. [18] *For any failure pattern with $f < P$, the number of block-steps required by the P-processor algorithm* W *with U leaves in the progress tree is $B = O\left(U + P\frac{\log P}{\log\log P}\right)$.*

Lemma 10. *For any failure pattern with $f \leq c\frac{P}{\log P}$ (for any $c > 0$), the number of block-steps required by the P-processor algorithm* W *with U leaves in the progress tree is $B = O(U + P \log_{\frac{P}{f}} P)$.*

Theorem 8. *Algorithm* W *solves Do-All(N, P, f) with work $O(N + \mathcal{Q}_{P,f} \log N)$.*

6.2 Iterative Do-All and Parallel Algorithm Simulations

We now consider the complexity of shared-memory *r-Do-All*(N, P, f) and of PRAM simulations.

Theorem 9. *The r-Do-All(N, P, f) problem can be solved on P crash-prone processors ($P \le N$), using shared memory, with work $W = O(r \cdot (N + \mathcal{R}_{r,P,f} \log N))$.*

Theorem 10. *Any synchronous N-processor, r-time shared-memory parallel algorithm (PRAM) can be simulated on P crash-prone synchronous processors with work $O(r \cdot (N + \mathcal{R}_{r,P,f} \log N))$.*

7 Conclusions

In this paper we gave the first complete analysis of the *Do-All* problem under the perfect load-balancing assumption. We introduced and analyzed the *iterative Do-All* problem that models repeated use of *Do-All* algorithms, such as found in algorithm simulations and transformations. A unique contribution of our analyses is that they precisely describe the effect of crash failures on the work of the computation. We demonstrate the utility of the analyses obtained with the perfect load-balancing assumption by using them to analyze message-passing and shared-memory algorithms and simulations that attempt to balance the loads among the processors.

References

1. Aumann, Y., Rabin, M.O.: Clock Construction in Fully Asynchronous Parallel Systems and PRAM Simulation. 33rd IEEE Symp. on Foundations of Computer Science (1993) 147–156.
2. Anderson, R.J., Woll, H.: Algorithms for the Certified Write All Problem. SIAM Journal of Computing, Vol. 26 **5** (1997) 1277–1283.
3. Buss, J., Kanellakis, P.C., Ragde, P., Shvartsman, A.A.: Parallel Algorithms with Processor Failures and Delays. Journal of Algorithms, Vol. 20 (1996) 45–86.
4. Chlebus, B.S., De Prisco, R., Shvartsman, A.A: Performing Tasks on Restartable Message-Passing Processors. Distributed Computing, Vol. 14 **1** (2001) 49–64.
5. Dasgupta, P., Kedem, Z., Rabin, M.: Parallel Processing on Networks of Workstation: A Fault-Tolerant, High Performance Approach. International Conference on Distributed Computer Systems (1995) 467–474.
6. De Prisco, R., Mayer, A., Yung, M.: Time-Optimal Message-Efficient Work Performance in the Presence of Faults. 13th ACM Symposium on Principles of Distributed Computing (1994) 161–172.
7. Dolev, S., Segala, R., Shvartsman, A.: Dynamic Load Balancing with Group Communication. 6th International Colloquium on Structural Information and Communication Complexity (1999) 111-125.
8. Dwork, C., Halpern, J., Waarts, O.: Performing Work Efficiently in the Presence of Faults. SIAM J. on Computing, Vol. 27 **5** (1998) 1457–1491.
9. Fischer, M.J., Lynch, N.A., Paterson, M.S.: Impossibility of Distributed Consensus with one Faulty Process. Journal of the ACM, Vol. 32 **2** (1985) 374–382.

10. Galil, Z., Mayer, A., Yung, M.: Resolving Message Complexity of Byzantine Agreement and Beyond. 36th IEEE Symp. on Foundations of Comp. Sc. (1995) 724–733.
11. Georgiou, C., Shvartsman, A.: Cooperative Computing with Fragmentable and Mergeable Groups. 7th International Colloquium on Structural Information and Communication Complexity (2000) 141–156.
12. Georgiou, C., Russell, A., Shvartsman, A.: The Complexity of Distributed Cooperation in the Presence of Failures. 4th International Conference on Principles of Distributed Systems (2000) 245–264.
13. Georgiou, C., Russell, A., Shvartsman, A.: The Complexity of Synchronous Iterative Do-All with Crashes. http://www.engr.uconn.edu/~acr/Papers/faults.ps.
14. Groote, J.F., Hesselink, W.H., Mauw, S., Vermeulen, R.: An Algorithm for the Asynchronous Write-All Problem Based on Process Collision. Distributed Computing (2001).
15. Hadzilacos, V., Toueg, S.: Fault-Tolerant Broadcasts and Related Problems. Distributed Computing, 2nd Ed., Addison-Wesley and ACM Press (1993).
16. Hesselink, W.H., Groote, J.F.: Waitfree Distributed Memory Management by Create, and Read Until Deletion (CRUD). Technical report SEN-R9811, CWI, Amsterdam (1998).
17. Kanellakis, P.C., Shvartsman, A.A.: Efficient Parallel Algorithms Can Be Made Robust. Distributed Computing, Vol. 5 (1992) 201–217.
18. Kanellakis, P.C., Shvartsman, A.A.: Fault-Tolerant Parallel Computation. Kluwer Academic Publishers (1997) ISBN 0-7923-9922-6.
19. Kedem, Z.M., Palem, K.V., Raghunathan, A., Spirakis, P.: Combining Tentative and Definite Executions for Dependable Parallel Computing. 23d ACM Symposium on Theory of Computing (1991) 381–390.
20. Kedem, Z.M., Palem, K.V., Rabin, M.O., Raghunathan, A.: Efficient Program Transformations for Resilient Parallel Computation via Randomization. 24th ACM Symp. on Theory of Computing (1992) 306–318.
21. Kedem, Z.M., Palem, K.V., Spirakis, P.: Efficient Robust Parallel Computations. 22nd ACM Symp. on Theory of Computing (1990) 138-148.
22. Lamport, L., Lynch, N.A.: Distributed Computing: Models and Methods. Handbook of Theoretical Computer Science, Vol. 1, North-Holland (1990).
23. Lamport, L., Shostak, R., Pease, M.: The Byzantine Generals Problem. ACM TOPLAS, Vol. 4 **3** (1982) 382–401.
24. Malewicz, G.G., Russell A., Shvartsman, A.A.: Distributed Cooperation in the Absence of Communication. 14th International Symposium on Distributed Computing (2000) 119–133.
25. Martel, C., Subramonian, R.: On the Complexity of Certified Write-All Algorithms. Journal of Algorithms, Vol. 16 **3** (1994) 361–387.
26. Martel, C., Park, A., Subramonian, R.: Work-Optimal Asynchronous Algorithms for Shared Memory Parallel Computers. SIAM Journal on Computing, Vol. 21 (1992) 1070–1099.
27. Martel, C., Subramonian, R., Park, A.: Asynchronous PRAMs are (Almost) as Good as Synchronous PRAMs. 32d IEEE Symp. on Foundations of Computer Science (1990) 590–599.
28. Pease, M., Shostak, R., Lamport, L.: Reaching Agreement in the Presence of Faults. Journal of the ACM, Vol. 27 **2** (1980) 228–234.
29. Shvartsman, A.A.: Achieving Optimal CRCW PRAM Fault-Tolerance. Information Processing Letters, Vol. 39 **2** (1991) 59–66.
30. Schlichting, R.D, Schneider, F.B.: Fail-Stop Processors: An Approach to Designing Fault-Tolerant Computing Systems. TOCS 1, Vol. 3 (1983) 222–238.