Distrib. Comput. (2004) 17: 47–63
Digital Object Identifier (DOI) 10.1007/s00446-003-0099-3

DISTRIBUTED COMPUTING

The complexity of synchronous iterative Do-All with crashes*

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

[1] Department of Computer Science and Engineering, University of Connecticut, 371 Fairfield Rd., Unit 1155, Storrs, CT 06269, USA (e-mail: {cg2, acr, aas}@cse.uconn.edu)
[2] Laboratory for Computer Science, Massachusetts Institute of Technology, 200 Technology Square, NE43-363, Cambridge, MA 02139, USA

Received: May 2002 / Accepted: June 2003
Published online: February 6, 2004 –

Abstract. The ability to cooperate on common tasks in a distributed setting is key to solving a broad range of computation problems ranging from distributed search such as SETI to distributed simulation and multi-agent collaboration. *Do-All*, an abstraction of such cooperative activity, is the problem of performing N tasks in a distributed system of P failure-prone processors. 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 all processors are counted. Work is ideally expressed as a function of N, P, **and** f, *the number of processor crashes*. However the known lower bounds and the upper bounds for extant algorithms do not adequately show how work depends on f. We present the first non-trivial lower bounds for *Do-All* that capture the dependence of work on N, P **and** f. 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 typical algorithm simulations. Our f-sensitive analysis enables us to derive tight bounds for *r-iterative Do-All* work (that are 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 under "free" load-balancing, 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. We give an improved analysis of an efficient message-passing algorithm. We also derive a tight and complete analysis of the best known *Do-All* algorithm for the synchronous shared-memory model. Finally we present a new upper bound on simulations of synchronous shared-memory algorithms on crash-prone processors.

Keywords: Fault-tolerance – Distributed algorithms – Work complexity – Lower bounds

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 is referred to as the *Do-All* problem:

P fault-prone processors perform N independent tasks,

and it is one of the standard problems in the research on the complexity of fault-tolerant distributed computation [9, 18]. Variations of this problem have been studied in shared-memory models (*Write-All*) [17,20,25], in message-passing models [7,9,11], and in partitionable networks (*Omni-Do*) [8, 14,23]. 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. Such tasks could be copying a large array, searching a collection of data, or applying a function to all elements of a matrix [15].

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 [27] (stop-failures are detectable, and in synchronous settings stop-failures are the same as crashes). The notion of *work complexity* reflects the *total* number of processing steps expended by an algorithm [17]. A distinguishing feature of our results is that the complexity is expressed in

* This work is supported in part by the NSF TOC Grants 9988304 and 0311368, and the NSF ITR Grant 0121277. The work of the second author is supported in part by the NSF CAREER Award 0093065. The work of the third author is supported in part by the NSF CAREER Award 9984778.

terms of the *number of processor crashes* f in addition to P and N.

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 is given as a function of N and P [18]. This is also the case with the best known asynchronous shared-memory algorithm [1]. Similarly, the best known lower bound for shared-memory models [20] and the best known lower bound applicable to message-passing models [3] do not involve f. The work of message-passing algorithms, e.g., [7,11], typically *does* include f, but this is due to the use of single coordinators, which means that for f coordinator failures the work necessarily includes an additive term $f \cdot P$. A message-passing algorithm using multiple coordinators [4] avoids this inefficiency and includes a term 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 synchronous shared-memory computing with failure-prone processors the impact of imprecise analysis of work complexity is especially significant. Approaches such as [21,28] use the *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 synchronous N-processor, τ-time 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 we shall see that 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 crashes. Let N be the size of the *Do-All* problem, P the number of processors, and f the number of crashes ($0 \leq f < P$). 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 ($f \leq P/\log(\min(N,P))$).

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 [18]). We also give an improved analysis of the work and message complexity for an efficient message-passing algorithm (algorithm AN [4]). Finally, our results yield insight about the bounds on task execution redundancy incurred when a central authority repeatedly allocates tasks to crash-prone processors.

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

Related work – algorithm simulations. Solutions for the *Do-All* problem can be used iteratively to simulate parallel algorithms formulated for synchronous failure-free processors in deterministic and probabilistic settings [21,20,24,26,28]. 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) each simulated processor is provided with a fixed number of auxiliary shared memory cells 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 [2,19]. Another important aspect of algorithm simulations is the use of an optimistic approach, where the computation may proceed for several steps assuming that all tasks assigned to active processors are successfully completed, e.g., [20]. In some deterministic models optimal simulations are possible (cf. [28]), however randomized solutions are able to achieve optimality (with high probability) for broader ranges of models and algorithms. An example of a practical implementation is discussed in [6].

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

Preliminary versions of the results in this paper appeared in extended abstracts [12,13].

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 oracle $\mathcal{O}$ that makes available to the processors a deterministic function of the global history of the computation: in particular, the oracle may be used to assist the processors to load-balance and terminate (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: *(i)* of course, any lower bound developed for this strong model applies equally well to message-passing or shared memory models as the oracle may make available to each processor the message or memory-read history it would have observed, *(ii)* any algorithm in the oracle model may be simulated in a message-passing or shared memory model by suitably simulating the oracle. As most *Do-All* algorithms use communication to load-balance and detect termination, this framework allows for the complexity analysis of specific algorithms to be divided into two parts: *(1)* the analysis of the cost of tolerating failures while performing work assuming perfect load-balancing, and *(2)* 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.

One may study the complexity of *Do-All*$^{\mathcal{O}}(\cdot)$ in conjunction with a variety of different oracles. When we prove lower bounds, we consider arbitrary oracles that may make available, for example, the entire global history of the computation on every processor. When we prove upper bounds, we use the *load-balancing* oracle, $\mathcal{O}_L$, described next. During each synchronous iteration of the computation, the oracle $\mathcal{O}_L$ makes available to each processor P_i two values: *Oracle-complete*, a Boolean which takes the value true if and only if all tasks were completed at the beginning of this iteration, and *Oracle-task*$_i$, an integer whose value is a task identifier with the property that the identifiers associated to the live processors are split evenly among the tasks. In particular, if processors $i_1, \ldots, i_k \in \{1, \ldots, P\}$ are alive and tasks $j_1, \ldots, j_\ell \in \{1, \ldots, N\}$ are incomplete at the beginning of the iteration, then *Oracle-task*$_{i_s} = j_t$, where $t = (s - 1 \bmod \ell) + 1$.

It was previously known that *Do-All*$^{\mathcal{O}}(N, P, f)$ can be solved with work $O(N+P \log P/ \log\log P)$ where $f < P \leq N$. A matching lower bound was also established for the specific case where $f = P/\log\log P+O(P/(\log\log P)^2))$ [18]. This meant that as long as the adversary can cause at least $P/\log\log P$ failures, *Do-All*$^{\mathcal{O}}(N, P, f)$ has matching upper and lower bounds of $\Theta(N + P \log P/ \log\log P)$. Finally, the upper bounds on work of the iterative use of *Do-All* were computed as the product $r \cdot W(N, P, f)$, where r is the number of iterations and $W(N, P, f)$ is the *Do-All* upper bound, even though fewer than f failures may be "available" to the adversary per iteration when $r > 1$.

Thus prior to our current results: (i) no non-trivial lower bounds were known for $f < P/\log\log P$, (ii) no f-sensitive analysis was available for the upper bounds in the range $1 < f < P/\log\log P$, and (iii) no precise iteration-sensitive bounds were known for *r-iterative Do-All*. Yet practical concerns would be well served by the knowledge of what happens in *Do-All* when the number of failures is moderate and when the *Do-All* algorithms are used iteratively. In particular, it is important to understand the behavior of the best algorithms for the entire range of failures f and iterations r.

The contributions in this work are as follows.[1]

I. We provide upper bounds (Sect. 4.1) and matching lower bounds (Sect. 4.2) that are f-sensitive, specifically we give a *complete* analysis of *Do-All*$^{\mathcal{O}}(N, P, f)$ for the *entire* range of f. The bounds on work $W(N, P, f)$ are:[2]

$$\begin{aligned}
&(a)\ W(N,P,f) = \Theta\left(N + P + P\tfrac{\log(\min(N,P))}{\log(P/f)}\right) \\
&\qquad \text{when } f \leq \tfrac{P}{\log(\min(N,P))}, \\
&(b)\ W(N,P,f) = \Theta\left(N + P + P\tfrac{\log(\min(N,P))}{\log\log(\min(N,P))}\right) \\
&\qquad \text{when } f > \tfrac{P}{\log(\min(N,P))}.
\end{aligned} \tag{1}$$

The lower bounds of course apply to algorithms in weaker models (e.g., a message passing or shared memory model).

In the rest of the paper we use a shorthand notation used to combine the statements such as (1)(a) and (1)(b) above. We define the quantity $\Lambda^r_{N,P,f}$, where r is the number of *Do-All* iterations (for *r-iterative Do-All*), and where N, P, and f appear in their usual roles, as follows.

$$\Lambda^r_{N,P,f} = \begin{cases} \log\left(\frac{Pr}{f}\right) & \text{when } f \leq \frac{Pr}{\log(\min(N,P))}, \\ \log\log(\min(N,P)) & \text{when } f > \frac{Pr}{\log(\min(N,P))}. \end{cases}$$

For a single instance *Do-All*, i.e., when $r = 1$, we simplify this by defining $\Lambda_{N,P,f} = \Lambda^1_{N,P,f}$. Thus the result (1) is succinctly given by the following.

$$W(N,P,f) = \Theta\big(N + P + P\log(\min(N,P))/\Lambda_{N,P,f}\big). \tag{2}$$

We use our bounds (2) (or equivalently (1)) to derive new bounds for algorithms where the extant analyses do not integrate f adequately. This is done by analyzing how load-balancing is implemented by the algorithms, e.g., by using coordinators or global data-structures. We show the following.

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

$$\begin{aligned}
W(N,P,f) &= O\left(\log f\left(N + P\log P/\Lambda_{N,P,f}\right)\right), \\
M(N,P,f) &= O\left(N + P\log P/\Lambda_{N,P,f} + Pf\right).
\end{aligned}$$

[1] In this paper, we define the asymptotic notation with multiple variables as follows: the expression $g(x, y, z) = O(f(x, y, z))$, for variables x, y, z taking values in $\mathbb{N}$, means that there exist positive constants b and c so that if $x, y, z \geq b$ then $g(x, y, z) \leq c \cdot f(x, y, z)$. The Θ and Ω notations are defined similarly.

All logarithms are to the base 2 unless explicitly specified otherwise. It is understood, that when $f = 0$, then $X/\log(Y/f) = 0$, for any $X \neq 0$ and $Y \neq 0$. Also, when $\log(X)$ appears in the results, it is understood that X is sufficiently large, so that $\log(X) \geq 1$ (especially since we are dealing with asymptotic analysis); note however that when we give the proofs of our results, we explicitly state the value of X.

[2] We use Θ notation to specify upper and lower bounds in conjunction with the work function $W(N, P, f)$ defined to be the minimum over all algorithms and oracles of the maximum work caused by all adversaries.

III. In Sect. 6.1 we give a *complete* analysis of the work complexity $W(N,P,f)$ of the algorithm of Kanellakis and Shvartsman [18] that solves the *Write-All* problem in synchronous shared-memory systems (shared memory *Do-All*) with processor crashes ($P \leq N$):

$$W(N,P,f) = O\left(N + P\log N \log P/\Lambda_{N,P,f}\right).$$

Note that the two algorithms [4,18] 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, and to demonstrate the utility and the generality of our approach. (We restrict our attention to the case $P \leq N$ for consistency with [4,18].)

Do-All algorithms have been used in developing *simulations* of failure-free algorithms on failure-prone processors, e.g., [21,28]. This is done by iteratively using a *Do-All* algorithm to simulate the steps of the N failure-free "virtual" processors on P failure-prone "physical" processors (here the usual case is that the number of physical processors does not exceed the number of virtual processors, i.e., $P \leq N$). In this paper we abstract this idea as the *iterative Do-All* problem as follows:

> The r-*iterative Do-All*problem, denoted r-*Do-All* (N,P,f), is the problem of using P processors to solve r instances of N-task *Do-All* with the added restriction that every task of the ith instance must be completed before any task of the $i+1$st instance is begun.

The oracle version r-*Do-All*$^{\mathcal{O}}(N,P,f)$ is defined analogously. 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 $r \cdot W(N,P,f)$. This does not take into account the fact that f crashes occur throughout the r iterations. We present a substantially better analysis of work, denoted $W(r,N,P,f)$, for crash-prone processors:

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

$$W(r,N,P,f) = \Theta\left(r \cdot \left(N + P\log P/\Lambda^r_{N,P,f}\right)\right). \quad (3)$$

The improvement in the bounds (3) are twofold. First, the derivation of the bounds reflects our improved failure-sensitive analysis for a single *Do-All*. Additionally, we have $\Lambda^r_{N,P,f} \geq \Lambda_{N,P,f}$ for any r, moreover $\Lambda_{N,P,f}$ is a constant with respect to r. The result is that our bounds (3) are asymptotically better than those obtained by computing the product of r and the (non-iterated) *Do-All* bounds (1) (for $P \leq N$).

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

$$W(r,N,P,f) = O\left(r \cdot \log\left(\frac{f}{r}\right) \cdot \left(N + P\log P/\Lambda^r_{N,P,f}\right)\right),$$
$$M(r,N,P,f) = O\left(r \cdot \left(N + P\log P/\Lambda^r_{N,P,f}\right) + fP\right).$$

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

$$W(r,N,P,f) = O\left(r \cdot \left(N + P\log N \log P/\Lambda^r_{N,P,f}\right)\right).$$

This last result shows a failure-sensitive improvement over the previously known deterministic bounds of $O(r \cdot (N + P\log N \log P/\log\log N))$ for deterministic parallel algorithm simulations using the *Write-All* algorithm [18] (the best known to date) together with the simulation techniques such as [21,28].

3 Models and definitions

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

Distributed setting. We consider a distributed system consisting of P processors; each processor has a unique processor identifier (PID) from the set $[P] = \{1,2,\ldots,P\}$. We assume that P is fixed and is known to all processors. Processors carry out their activities by executing *steps*. The exact nature of a step depends on the model of computation, and specifically on whether communication is by message-passing or accessing shared memory, as discussed later in this section under *Communication*. Note that the upper and lower bounds presented in Sect. 4 are abstracted away from any communication concerns.

Tasks. We define a *task* to be a computation that can be performed by any processor in at most one step; its execution does not depend 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. The set $\mathcal{T}$ of TIDs is totally ordered. When we consider the *Do-All* problem, we simply assume that $\mathcal{T} = [N]$. We shall also consider sequences of task-sets $\mathcal{T}_1,\ldots,\mathcal{T}_r$, where each $\mathcal{T}_i$, for $1 \leq i \leq r$, is a set of N totally ordered tasks, and the execution of any task in $\mathcal{T}_i$ must be delayed until all tasks in $\mathcal{T}_{i-1}$ are complete. In all cases we assume that the tasks are initially known to all the processors.

Model of failures and synchrony. We assume the *fail-stop* processor model [27]. A processor may crash at any moment during the computation (including within a step) and once crashed it does not restart. The processors are synchronous. This means that the activity of processors is governed by a common clock. A processor that has not crashed takes one clock tick to execute one step (recall that the precise activities within a step depend on the specific model of computation). The steps are sequentially enumerated using natural numbers according to the common clock.

We let an omniscient *adversary* determine when to impose crashes, and we use the term *failure pattern* to denote the set of the events, i.e., crashes, caused by the adversary. Following [17], we define a failure pattern F as a set of triples $\langle$*crash*, PID, $t\rangle$ where crash is the event caused by the adversary, PID is the identifier of the processor that crashes, and t is the step of the computation in which the adversary forced processor PID to crash. We require that any failure pattern F contain at most one triple $\langle$*crash*, PID, $t\rangle$ for any PID, i.e., if processor PID crashes, the step t during which it crashes is uniquely defined.

When a computation occurs in the presence of a failure pattern F, we say that processor PID $\in [P]$ *survives* step i, if for all steps $j \leq i$, $\langle$*crash*, PID, $j\rangle \notin F$.

For a failure pattern F, we define its *size* $|F|$ to be the number of crashes. For the purpose of this paper we consider only the failure patterns with $|F| < P$, that is we require that the adversary leaves at least one processor operational to ensure computational progress. Then, we define the *failure model* $\mathcal{F}$ to be the set of all failure patterns F with $|F| < P$. We assume that the processors have neither knowledge of F nor of any bounds on $|F|$. Indeed, the algorithms discussed in this paper are correct for any failure pattern $F \in \mathcal{F}$.

The Oracle model. In Sect. 4 we consider computation where processors are assisted by a deterministic omniscient *oracle*. The introduction of the oracle serves two purposes.

1. The oracle strengthens the model by providing the processors with, for example, load-balancing and termination information. For our upper bounds, we shall concentrate on the *load-balancing* oracle $\mathcal{O}_L$ (discussed in Sect. 2): it informs the processors whether or not the computation is completed via the boolean *Oracle-complete*, and if not, what task to perform next via the integer value *Oracle-task*. For our lower bounds, the processors may use the oracle to glean *any* information about the history of the computation. Of course, the lower bounds established for the oracle model also apply to any weaker model.
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.

For convenience, when we consider *Do-All* algorithms where processors are permitted to interact with an oracle $\mathcal{O}$, we consider the computation to be divided into a sequence of *iterations*. Each iteration consists of two phases. In the first phase, each live processor may query the oracle any number of times and, based on this information, must select a task to complete next. In the second phase, each processor completes the task it has just selected. As we wish to prove lower bounds for the strongest possible model, we do not place any restriction on the computation which takes place during the first phase; indeed, when we measure work, we only count processing dedicated to completing tasks. On the other hand, upper bounds which use the load-balancing oracle $\mathcal{O}_L$ actually perform only $O(1)$ computation during this first phase.

Communication. In Sects. 5 and 6 we deal with more specific message-passing and shared-memory models of computation. For computation in the message-passing model, we assume that there is a known constant upper bound on message delays. Specifically, within a step, a processor can send messages to other processors and receive messages from other processors sent to it in the previous step. A crash may occur at any point during a step. Any sends and receives preceding the crash complete correctly (and no sends or receives following the crash occur). We further assume that when a processor explicitly *multicasts* a message to a group of processors then the multicast is reliable [16]: either all messages are delivered to (non-crashed) processors, or no messages are delivered. (Communication complexity is defined in Sect. 5.)

When considering computation in the shared-memory model (Sect. 6) we assume that reading or writing to a memory cell takes one time unit. Thus a constant number of reads and writes can take place during a step. Reads and writes can be concurrent; when two or more processors simultaneously write to the same memory cell, either *common* or *arbitrary* concurrent write discipline is observed. This follows the conventions established for the Parallel Random Access Machine (PRAM) [10]: for the common writes it is assumed that all values concurrently written to a memory location are the same, and for the arbitrary writes it is assumed that the concurrent writes to the memory location are arbitrarily ordered. Finally, a crash can occur at any point during a step. Any reads and writes preceding the crash complete correctly (and no reads or writes following the crash occur).

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

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

We let *Do-All*(N, P, f) stand for the *Do-All* problem for N tasks, P processors, 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 $\mathcal{O}$. In this work we consider an instance of the *Do-All* problem to be solved when all N tasks have been performed and when at least one processor knows this.

We define the *iterative* variation of the *Do-All* problem as follows:

> *Iterative Do-All: Given a sequence $\mathcal{T}_1, \ldots, \mathcal{T}_r$ of r sets of N tasks, perform all $r \cdot N$ tasks using P processors, in the presence of any failure pattern F in the failure model $\mathcal{F}$, with the added restriction that every task in $\mathcal{T}_i$ must be completed before any task in $\mathcal{T}_{i+1}$ is begun.*

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.

In the shared-memory model the *Do-All* problem is known as the *Write-All* problem. The main difference is that in *Do-All* the tasks may be supplied to the processors from some external sources, while in *Write-All* the tasks are stored in shared-memory accessible to all processors. In the context of this work we abstract away from the sources and the nature of the tasks and we treat *Do-All* and *Write-All* as the same problem. We refer to *Write-All* explicitly only when we apply our results in the shared-memory model.

Measuring efficiency. We are interested in studying the complexity of *Do-All* measured as *work* (cf. [18,9,7]). 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, thus performing a task corresponds to a unit of work). Our definition of *work complexity* is based on the *available processor steps* measure [17]. Let $\mathcal{F}$ be the failure model. For a computation subject to a failure pattern $F \in \mathcal{F}$, denote by $P_i(F)$ the number of processors surviving 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 step $\tau(F)$, then the work complexity W of the algorithm is:*

$$W_{N,P,f} = \max_{F \in \mathcal{F},\ |F| \leq f} \left\{ \sum_{i \leq \tau(F)} P_i(F) \right\}.$$

Note that idling processors that survive a step are charged for a unit of work 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 Sect. 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 ($f < P$).

4.1 Do-All upper bounds

To study the upper bounds for *Do-All* we give an oracle-based algorithm in Fig. 1. The oracle $\mathcal{O}_L$ (see Sect. 2) performs the termination and load-balancing computation on behalf of the processors.

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

Fig. 1. Oracle-based algorithm

Lemma 1. [17] *The Do-All$^{\mathcal{O}_L}(N, P, f)$ problem can be solved for any pattern F of crashes with $|F| \leq f < P$ and $P \leq N$ using work*

$$W = O\left(N + P\frac{\log P}{\log\log P}\right).$$

Lemma 2. *The Do-All$^{\mathcal{O}_L}(N, P, f)$ problem can be solved for any pattern F of crashes with $|F| \leq f < P$ and $P \geq N$ using work*

$$W = O\left(P\frac{\log N}{\log\log N}\right).$$

Proof. Let $\rho = P/N$, and for simplicity assume that $P \bmod N = 0$. We first divide the P processors into N groups of processors, each group consisting of ρ processors. We then assign the same processor id to all the processors in the same group. Due to the synchrony of the system, the processors with the same id will exhibit the same behavior under the oracle $\mathcal{O}_L$.[3] So, we view each group of processors as a *virtual processor* with an id taken from the set $[N]$. Each virtual processor represents ρ physical processors. By doing that, we translate the problem into one where N processors need to perform N tasks. Hence, by applying Lemma 1 to this new version of the problem, we see that the total work is

$$\begin{aligned} W &= \rho \cdot O(N + N\log N/\log\log N) \\ &= O(\rho N + \rho N \log N/\log\log N). \end{aligned}$$

The result now follows from the fact that $\rho N = P$. □

Note that Lemmas 1 and 2 do not show how, if at all, work depends on f.

Lemma 3. *The Do-All$^{\mathcal{O}_L}(N, P, f)$ problem can be solved for any pattern of crashes with $f \leq P/\log(\min(N, P))$, using work*

$$W = O\left(N + P + P\log_{\frac{P}{f}}(\min(N, P))\right).$$

Proof. For an *iteration* of the oracle-based algorithm (Fig. 1), let Δf denote the number of processor crashes in this iteration. (Δf can be different for each iteration, though the sum of these for all iterations cannot exceed f.) We set $b = b(P, f) = \frac{P}{2f}$, and we define $W(N, P, f)$ to be the work required to solve *Do-All*$^{\mathcal{O}_L}(N, P, f)$. Our goal is to show that for all U, P and f, the work $W(U, P, f)$ is no more than $16P + U + P\log_{\frac{P}{2f}}(\min(U, P))(= O(P + U + P\log_{\frac{P}{f}}(\min(U, P))))$, where $U \leq N$ denotes the number of undone tasks. The proof proceeds by induction on U.

Base Case: Observe that when $U \leq 16$, $W(U, P, f) \leq 16P < 16P + U + P\log_b(\min(U, P))$, for all P and f.

Inductive Hypothesis: Assume that we have proved the theorem for all $U < \hat{U}$ ($\hat{U} \leq N$) and all P and f.

Inductive Step: Consider $U = \hat{U}$. We investigate two cases:

Case 1: $P \leq \hat{U}$ (in particular, $\min(\hat{U}, P) = P$). In this case each processor is assigned to a unique task, hence

$$W(\hat{U}, P, f) \leq P + \max_{0 \leq \Delta f \leq f} W(\hat{U} - P + \Delta f, P - \Delta f, f - \Delta f).$$

As $P - \Delta f > 0$, $\hat{U} - P + \Delta f < \hat{U}$ and, by the induction hypothesis,

$$W(\hat{U}, P, f) \leq P + \max_{0 \leq \Delta f \leq f} \Big[16(P - \Delta f) + (\hat{U} - P + \Delta f) + (P - \Delta f)\log_{b(P-\Delta f, f-\Delta f)}(\min(\hat{U} - P + \Delta f, P - \Delta f)\Big]$$

[3] Alternatively, one can modify oracle $\mathcal{O}_L$ so that it divides processors into groups by treating the processors in the same group as if they had the same id.

Now, $b(P-\Delta f, f-\Delta f) \geq b(P,f)$, and

$$\log_{b(P,f)}(\min(\hat{U}-P+\Delta f, P-\Delta f) \leq \log_{b(P,f)}(P-\Delta f),$$

so that

$$\begin{aligned} W(\hat{U},P,f) &\leq 16P + \hat{U} + P\log_{b(P,f)} P \\ &= 16P + \hat{U} + P\log_{b(P,f)}(\min(\hat{U},P)), \end{aligned}$$

as desired.

Case 2: $P > \hat{U}$ (in particular, $\min(\hat{U},P) = \hat{U}$). In this case, by assumption we have

$$W(\hat{U},P,f) \leq P + \max_{0\leq\Delta f\leq f} W(\gamma\hat{U}, P-\Delta f, f-\Delta f),$$

where $\gamma = \gamma(\hat{U},P,\Delta f)$ is the ratio of the number of the remaining tasks to $\hat{U}$ ($0 \leq \gamma < 1$).
Let $\phi = \Delta f/P \leq f/P < 1$, the fraction of processors which fail during this iteration; then $\phi/2 < \gamma < 2\phi$. (To see this, observe that

$$\frac{\phi P}{\lceil P/\hat{U}\rceil\hat{U}} = \frac{\phi P/\lceil P/\hat{U}\rceil}{\hat{U}} \leq \gamma \leq \frac{\phi P/\lfloor P/\hat{U}\rfloor}{\hat{U}} = \frac{\phi P}{\lfloor P/\hat{U}\rfloor\hat{U}}.$$

Let $P = c\hat{U}$, $c > 1$. Then

$$\frac{c}{\lceil c\rceil}\phi = \frac{\phi c\hat{U}}{\lceil c\rceil\hat{U}} \leq \gamma \leq \frac{\phi c\hat{U}}{\lfloor c\rfloor\hat{U}} = \frac{c}{\lfloor c\rfloor}\phi.$$

Now observe that $1 \leq \frac{c}{\lfloor c\rfloor} < 2$ and $1/2 < \frac{c}{\lceil c\rceil} \leq 1$, $\forall c > 1$, and hence, $\phi/2 < \gamma < 2\phi$, as desired.) Then,

$$W(\hat{U},P,f) \leq P + \max_{\phi\in[0,f/P]} W(\gamma\hat{U}, (1-\phi)P, f-\phi P).$$

As $\gamma\hat{U} < \hat{U}$, we may apply the induction hypothesis:

$$\begin{aligned} W(\hat{U},P,f) \leq P + \max_{\phi\in[0,f/P]} \Big[& 16(1-\phi)P \\ & + \gamma\hat{U} + (1-\phi)P\log_{b'}(\min(\gamma\hat{U},(1-\phi)P)) \Big], \end{aligned}$$

where $b' = b(P-\phi P, f-\phi P)$. As above, $b' \geq b(P,f)$ and $\min(\gamma\hat{U},(1-\phi)P)) \leq \gamma\hat{U}$, so that

$$\begin{aligned} W(\hat{U},P,f) \leq P + \max_{\phi\in[0,f/P]} \Big[& 16(1-\phi)P \\ & + \gamma\hat{U} + (1-\phi)P\log_{b(P,f)}(\gamma\hat{U}) \Big]. \end{aligned}$$

To complete the proof, it suffices to show that for all $\phi \in [0, f/P]$, $15P + P\log_{b(P,f)}\hat{U} - (1-\phi)P\log_{b(P,f)}(\gamma\hat{U}) \geq 16(1-\phi)P - \hat{U}(1-\gamma)$.
Upper bounding $16(1-\phi)P - \hat{U}(1-\gamma)$ with $16(1-\phi)P$ and dividing through by P, it is sufficient to show that

$$15 + \log_{b(P,f)}\hat{U} - (1-\phi)\log_{b(P,f)}(\gamma\hat{U}) \geq 16(1-\phi),$$

or, equivalently,

$$\log_{b(P,f)}\hat{U} - (1-\phi)\log_{b(P,f)}(\gamma\hat{U}) \geq 1 - 16\phi.$$

We now focus on the left hand side of the above equation:

$$\begin{aligned} \log_{b(P,f)}\hat{U} - (1-\phi)\left[\log_{b(P,f)}\gamma + \log_{b(P,f)}\hat{U}\right] = \\ \phi\log_{b(P,f)}\hat{U} + (1-\phi)\log_{b(P,f)}\gamma^{-1}. \end{aligned}$$

Since $f \leq \frac{P}{\log(\min(\hat{U},P))} = \frac{P}{\log\hat{U}}$, for any $\hat{U} > 16$ we have that $\frac{P}{2f} > 2$. Observe that,

$$\phi\log_{b(P,f)}\hat{U} + (1-\phi)\log_{b(P,f)}\gamma^{-1} \geq (1-\phi)\log_{b(P,f)}\gamma^{-1}$$

since $\hat{U} \geq P/f > P/2f$. (Note that if $\hat{U} < P/f$, then all tasks are completed in this iteration.)
Recall that $\gamma^{-1} \geq (2\phi)^{-1}$ and $\phi < f/P$. Therefore,

$$(1-\phi)\log_{b(P,f)}\gamma^{-1} \geq (1-\phi)\log_{b(P,f)}(2\phi)^{-1} \geq 1 - 16\phi.$$

Evidently,

$$W = O(N + P + P\log_{\frac{P}{f}}(\min(N,P))),$$

as desired. □

Recall that $\Lambda_{N,P,f}$ is $\log(P/f)$ when

$$f \leq P/\log(\min(N,P))$$

and $\log\log(\min(N,P))$ when

$$f > P/\log(\min(N,P)).$$

We now give our main upper-bound result.

Theorem 1. *Do-All$^{\mathcal{O}_L}(N,P,f)$ can be solved for any pattern of crashes using work*

$$W = O\left(N + P + P\log(\min(N,P))/\Lambda_{N,P,f}\right).$$

Proof. This follows from the definition of $\Lambda_{N,P,f}$ and Lemmas 1, 2 and 3. □

4.2 Do-All lower bounds

We now develop the lower bounds for *Do-All*$^{\mathcal{O}}(N,P,f)$; these bounds match the upper bounds presented in Sect. 4.1. Note that the results in this section hold also for the *Do-All*(N,P,f) problem (without the oracle).

The following mathematical facts (from [18]) are used in the proofs.

Fact 1 *If $a_1, a_2, \ldots, a_m$ $(m > 1)$ is a sorted list of non-negative integers, then for all j $(1 \leq j < m)$ we have $\left(1 - \frac{j}{m}\right)\sum_{i=1}^{m} a_i \leq \sum_{i=j+1}^{m} a_i$.*

Fact 2 *Given $N \in \mathbb{N}$, $\kappa \in \mathbb{R}$, such that $N \cdot \kappa > 1$, $\kappa \leq \frac{1}{2}$, and $\sigma \in \mathbb{N}$ such that $\sigma < \frac{\log N}{\log(\kappa^{-1})} - 1$, then the following inequality holds: $\lfloor \ldots \lfloor \lfloor N \underbrace{\cdot \kappa \rfloor \cdot \kappa \rfloor \ldots \cdot \kappa \rfloor}_{\sigma \text{ times}} > 0$.*

Proof. To show the result it suffices to show that, after dropping one floor and strengthening the inequality:

$$(\lfloor \ldots \lfloor \lfloor N \underbrace{\cdot \kappa \rfloor \cdot \kappa \rfloor \ldots \cdot \kappa \rfloor}_{\sigma-1 \text{ times}} \cdot \kappa) - 1 > 0\,,$$

or that

$$\lfloor \ldots \lfloor \lfloor N \underbrace{\cdot \kappa \rfloor \cdot \kappa \rfloor \ldots \cdot \kappa \rfloor}_{\sigma-1 \text{ times}} > \frac{1}{\kappa}\,.$$

Applying this transformation for $\sigma - 1$ more steps, we see that it suffices to show that $N > \frac{1}{\kappa^\sigma} + \frac{1}{\kappa^{\sigma-1}} + \ldots + \frac{1}{\kappa}$, or, using geometric progression summation, that

$$N > \frac{(\kappa^{-1})^{\sigma+1} - (\kappa^{-1})}{(\kappa^{-1}) - 1}\,.$$

We observe that

$$(\kappa^{-1})^{\sigma+1} > \frac{(\kappa^{-1})^{\sigma+1} - (\kappa^{-1})}{(\kappa^{-1}) - 1}$$

for $\kappa \leq \frac{1}{2}$, thus it is enough to show that $N > (\kappa^{-1})^{\sigma+1}$. After taking logarithms of both sides of the inequality, $\log N > (\sigma+1)\log(\kappa^{-1})$, and so it suffices to have $\sigma < \frac{\log N}{\log(\kappa^{-1})} - 1$. □

We now define a specific adversarial strategy used to derive our lower bounds. Let A be an iterative algorithm that solves the *Do-All* problem. Let P_i be the number of processors remaining at the end of the i^{th} iteration of A and let U_i denote the number of tasks that remain to be done at the end of iteration i. Initially, $P_0 = P \geq N = U_0$. The strategy of the adversary is defined for each iteration of the algorithm. Based on a variable κ, defined in the interval $(0, 1/2)$, the adversary determines which processors will be allowed to work and which will be stopped in a given iteration. We call this adversary $\mathfrak{A}$.

Adversary $\mathfrak{A}$:

Iteration 1: The adversary chooses $U_1 = \lfloor \kappa U_0 \rfloor$ tasks with the least number of processors assigned to them. This can be done since the adversary is omniscient; it knows all the actions to be performed by A (as well as any advice provided by the oracle). The adversary then crashes the processors assigned to these tasks, if any.

Iteration i: Among U_{i-1} tasks remaining after the iteration $i-1$, 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.

Termination: 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 A terminates.

We now study the adversarial strategy $\mathfrak{A}$ and derive lower bound results.

Remark 1 Relationship between N and κ: If κ is chosen so that $\kappa \cdot N \leq 1$ then by the strategy of $\mathfrak{A}$, an algorithm solving *Do-All* may be able to solve it in a constant number of iterations (namely two) with work $O(P)$. This is because $U_1 = \lfloor \kappa U_0 \rfloor \leq \kappa N \leq 1$. Henceforth we consider κ to be such that $\kappa \cdot N > 1$. □

Lemma 4. *For adversary* $\mathfrak{A}$*, if at the iteration* i *the number of remaining tasks is* $U_{i-1} > 1$*, then*

(a) $U_i = \lfloor \ldots \lfloor \lfloor N \underbrace{\cdot \kappa \rfloor \cdot \kappa \rfloor \ldots \cdot \kappa \rfloor}_{i \text{ times}}$*, and*

(b) $P_i \geq (1-\kappa)^i P_0$.

Proof. Part (a) is immediate from the definition of $\mathfrak{A}$. To express the number of surviving processors P_i for part (b), we use Fact 1 with the following definitions:

Let $m = U_{i-1}$, and let $a_1, \ldots, a_m$ be the quantities of processors assigned to each task, sorted in ascending order. Let a_m also include the quantity of any un-assigned processors, i.e., a_1 is the least number of processors assigned to a task, a_2 is the next least quantity of processors, etc. (In other words, $a_1 \leq a_2 \leq \ldots \leq a_m$.) Let $j = U_i$. Thus the adversary stops exactly $\sum_{i=1}^{j} a_i$ processors. At the beginning of iteration i, the number of processors $P_{i-1} = \sum_{i=1}^{m} a_i$, therefore, the number of surviving processors $P_i = \sum_{i=j+1}^{m} a_i$.

Using Fact 1, we have $P_i \geq \left(1 - \frac{U_i}{U_{i-1}}\right) P_{i-1}$, and after substituting for $U_i = \lfloor \kappa U_{i-1} \rfloor$ we have

$$P_i \geq \left(1 - \frac{\lfloor \kappa U_{i-1} \rfloor}{U_{i-1}}\right) P_{i-1} \geq (1-\kappa)\, P_{i-1} \geq (1-\kappa)^i P_0,$$

as desired. □

Lemma 5. *Given any algorithm solving the Do-All*$^{\mathcal{O}}$ (N, P, f) *problem under any oracle* $\mathcal{O}$*, adversary* $\mathfrak{A}$ *will cause the algorithm to cycle through at least* $\frac{\log N}{\log(\kappa^{-1})} - 1$ *iterations.*

Proof. Let τ be the earliest iteration when the last task is performed. We use Fact 2 with σ the largest integer such that $\sigma < \log N / \log(\kappa^{-1}) - 1$. Then $U_\sigma = \lfloor \ldots \lfloor \lfloor N \underbrace{\cdot \kappa \rfloor \cdot \kappa \rfloor \ldots \cdot \kappa \rfloor}_{\sigma \text{ times}} > 0$, and so τ must be greater than σ because $U_\tau = 0$. Thus, $\tau \geq \frac{\log N}{\log(\kappa^{-1})} - 1 > \sigma$. □

Lemma 6. *Given any algorithm* A *that solves Do-All*$^{\mathcal{O}}$ (N, P, f) *under any oracle* $\mathcal{O}$*, with* $P \geq N$ *and* $f < P$*, then adversary* $\mathfrak{A}$ *with* $\kappa = \frac{1}{\log N}$ *causes work*

$$W = \Omega\left(P \frac{\log N}{\log \log N}\right).$$

Proof. We first assume that $N > 4$ (we aim to establish an asymptotic result, and this eliminates uninteresting cases). Since $\kappa = 1/\log N$, we have that $\kappa \in (0, 1/2)$ when $N > 4$. From Lemma 4(a) and Lemma 5 we see that $\mathfrak{A}$ will cause algorithm A to iterate at least $\tau = (\log N / \log \log N) - 1$ times. Now observe that the work must be at least $P_\tau \cdot \tau$, where P_τ is the number of surviving processors after A terminates. From

Lemma 4(b) we have that $P_\tau \geq (1-\kappa)^\tau P_0 = (1-\frac{1}{\log N})^\tau P$. Therefore,

$$P_\tau \geq P\left(1-\frac{1}{\log N}\right)^{\frac{\log N}{\log\log N}-1} \geq P\left(1-\frac{1}{\log N}\right)^{\frac{\log N}{\log\log N}}$$
$$\geq P\left(1-\left(\frac{1}{\log N}\right)\cdot\left(\frac{\log N}{\log\log N}\right)\right) = P-\frac{P}{\log\log N}.$$

Let f_τ denote the actual number of crashes caused by the adversary. Then, $f_\tau = P - P_\tau \leq P - P + \frac{P}{\log\log N} = \frac{P}{\log\log N} < P$. Hence, $\mathfrak{A}$ when using this specific κ does not exceed the allowed number of crashes. Now, the work caused by $\mathfrak{A}$ is:

$$W = \Omega(P_\tau\cdot\tau) = \Omega\left(\left(P-\frac{P}{\log\log N}\right)\cdot\left(\frac{\log N}{\log\log N}-1\right)\right)$$
$$= \Omega\left(P\frac{\log N}{\log\log N}\right).$$

This completes the proof. □

Note that the above lower bound result generalizes the result given in [17,12]. There, the bound was given for the special case of $P = N$.

Corollary 1. *Given any algorithm* A *that solves Do-All*$^{\mathcal{O}}$ (N,P,f) *under any oracle* $\mathcal{O}$*, with* $f < P \leq N$*, then there exists an adversary that causes work*

$$W = \Omega\left(N + P\frac{\log P}{\log\log P}\right).$$

Proof. Note that $W = \Omega(N)$ because all tasks must be performed. From Lemma 6 we know that *Do-All*$^{\mathcal{O}}(P,P,f)$ requires $\Omega(P\log P/\log\log P)$ work. Given that work is nondecreasing in N (as follows from Definition 1) we obtain the desired result by combining the two bounds. □

Observe that Lemma 6 and Corollary 1 do not show how, if at all, work depends on f.

Lemma 7. *Given any algorithm* A *that solves Do-All*$^{\mathcal{O}}$ (N,P,f) *under any oracle* $\mathcal{O}$*, with* $P \geq N$*, then adversary* $\mathfrak{A}$ *with* $(\kappa^{-1})\log(\kappa^{-1}) = \frac{P\log N}{f}$ *and* $f \leq \frac{P}{\log N}$ *causes work*

$$W = \Omega\left(P + P\log_{\frac{P}{f}} N\right).$$

Proof. We assume that $N > 4$ (we aim to establish an asymptotic result, and this eliminates uninteresting cases). From $(\kappa^{-1})\log(\kappa^{-1}) = \frac{P\log N}{f}$, $f \leq \frac{P}{\log N}$, and $N > 4$ we see that $\log(\kappa^{-1}) > 4\kappa$. This implies that $\kappa \in (0,1/2)$. Hence, from Lemma 5 we have that $\mathfrak{A}$ will cause algorithm A to iterate at least $\tau = (\log N/\log(\kappa^{-1})) - 1$ times.

Now observe that the work must be at least $P_\tau\cdot\tau$, where P_τ is the number of surviving processors after A terminates. Recall from Lemma 4(b) that $P_\tau \geq (1-\kappa)^\tau P_0$. Therefore,

$$P_\tau \geq P(1-\kappa)^\tau \geq P(1-\kappa)^{\frac{\log N}{\log(\kappa^{-1})}-1} \geq P(1-\kappa)^{\frac{\log N}{\log(\kappa^{-1})}}$$
$$\geq P\left(1-\kappa\cdot\frac{\log N}{\log(\kappa^{-1})}\right) = P\left(1-\left(\frac{\kappa}{\log(\kappa^{-1})}\right)\log N\right)$$
$$= P\left(1-\left(\frac{f}{P\log N}\right)\log N\right) = P-f.$$

Let f_τ denote the actual number of crashes caused by the adversary. Then, $f_\tau = P - P_\tau \leq P - (P-f) = f$. Hence, $\mathfrak{A}$ when using this specific κ does not exceed the allowed number of crashes ($f \leq P/\log N$).

Recall that $(\kappa^{-1})\log(\kappa^{-1}) = \frac{P\log N}{f}$, therefore, $(\kappa^{-1}) = \Theta\left(\frac{\frac{P\log N}{f}}{\log(\frac{P\log N}{f})}\right)$. From this we obtain

$$\log(\kappa^{-1}) = \Theta\left(\log\left(\frac{P\log N}{f}\right) - \log\log\left(\frac{P\log N}{f}\right)\right)$$
$$= \Theta\left(\log\left(\frac{P\log N}{f}\right)\right).$$

Then, noting that $P_\tau \geq P - f \geq P - P/\log N = \Theta(P)$ and that $\kappa\cdot N > 1$ (see Remark 1), we assess the work W caused by $\mathfrak{A}$ as follows:

$$W = \Omega(P_\tau\cdot\tau) = \Omega\left(P\cdot\frac{\log N}{\log(\kappa^{-1})}\right)$$
$$= \Omega\left(P + P\frac{\log N}{\log(\frac{P\log N}{f})}\right).$$

Now recall that $P/f \geq \log N$. Hence, for any $N > 4$ we have that $P/f > 2$ and that $\log((P\log N)/f) = \log(P/f) + \log\log N = \Theta(\log(P/f))$. From the above,

$$W = \Omega\left(P + P\frac{\log N}{\log(\frac{P}{f})}\right) = \Omega\left(P + P\log_{\frac{P}{f}} N\right).$$

This completes the proof. □

Corollary 2. *Given any algorithm* A *that solves Do-All*$^{\mathcal{O}}$ (N,P,f) *for* $P \leq N$ *and under any oracle* $\mathcal{O}$*, there exists an adversary that causes* $f \leq \frac{P}{\log P}$ *crashes, and work*

$$W = \Omega\left(N + P\log_{\frac{P}{f}} P\right).$$

Proof. Note that $W = \Omega(N)$ because all tasks must be performed. From Lemma 7 we know that *Do-All*$^{\mathcal{O}}(P,P,f)$ requires $\Omega(P\log_{\frac{P}{f}} P)$ work, for $f \leq P/\log P$. Given that work is nondecreasing in N we obtain the desired result by combining the two bounds. □

We now give our main lower-bound result.

Theorem 2. *Given any algorithm* A *that solves Do-All*$^{\mathcal{O}}$ (N,P,f) *under any oracle* $\mathcal{O}$*, there exists an adversary that causes work*

$$W = \Omega\left(N + P + P\log(\min(N,P))/\Lambda_{N,P,f}\right).$$

Proof. We consider two cases.

Case 1: $P \leq N$.

For the range of failures $f \leq P/\log P$, per Corollary 2, the work is $\Omega(N + P\log_{P/f} P)$. From Corollary 2 we also obtain the fact that when $f = P/\log P$ then work must be $\Omega\left(N + P\log P/\log\log P\right)$. Note that this is the worst case work for any f (see Corollary 1). Therefore, for the range

$P/\log P < f < P$, the adversary establishes this worst case work using the initial $P/\log P$ failures.

Case 2: $P \geq N$.
For the range of failures $f \leq P/\log N$, per Lemma 7, the work is $\Omega(P + P\log_{P/f} N)$. From Lemma 7 we also obtain the fact that when $f = P/\log N$ then work must be $\Omega\left(P\log N/\log\log N\right)$. Note that this is the worst case work for any f (see Lemma 6). Therefore, for the range $P/\log N < f < P$, the adversary establishes this worst case work using the initial $P/\log N$ failures. The result now follows by combining the two cases and the definition of $\Lambda_{N,P,f}$. □

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 studying simulations, a *Do-All*(N, P, f) solution abstracts the setting where P physical crash-prone processors simulate N virtual processors, such that each task i among the N tasks in *Do-All* represents a single step of the virtual processor i. In this setting it is customary to consider the case with $P \leq N$, that is when the number of physical processors does not exceed the number of virtual processors. The *iterative Do-All* then models the simulation of multiple steps of the virtual processors. In this section we therefore assume that $P \leq N$. (We note however, that it is possible to extend the results for *iterative Do-All* for the cases when $P > N$, using similar arguments as in the case of $P \leq N$.)

In principle r-*Do-All*(N, P, f) can be solved by running an algorithm for *Do-All*(N, P, f) r times. For example, r-*Do-All*$^{\mathcal{O}}(N, P, f)$ can be solved by running the oracle-based algorithm in Fig. 1 in r iterations. 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 that takes into account the diminishing number of failures "available" to the adversary. We refer to each *Do-All* iteration as a *round* of r-*Do-All*$^{\mathcal{O}}(N, P, f)$.

Theorem 3. *The* r-*Do-All*$^{\mathcal{O}_L}(N, P, f)$ *problem for* $P \leq N$ *can be solved with work*

$$W = O\left(r \cdot \left(N + P\log P/\Lambda^r_{N,P,f}\right)\right)$$

Proof. Let r_i denote the i^{th} round of the iterative *Do-All*. Let P_i be the number of active processors at the beginning of r_i and f_i be the number of crashes during r_i. Note that $P_1 = P$, where r_1 is the first round of r-*Do-All*$^{\mathcal{O}_L}(N, P, f)$ and that $P_i \leq P$. We consider two cases:

Case 1: $f > \frac{Pr}{\log P}$. Consider a round r_i. From Theorem 1 we see that the work for this round is $O\left(N + P_i\log_{P_i/f_i} P_i\right)$ when $f_i \leq P_i/\log P_i$ and $O\left(N + P_i\log P_i/\log\log P_i\right)$ otherwise. However in this case, we can have $f_i = \Theta\left(P/\log P\right)$ for all r_i without "running out" of processors. Thus,

$$W_1 = O\left(r \cdot \left(N + P\frac{\log P}{\log\log P}\right)\right).$$

Case 2: $f \leq \frac{Pr}{\log P}$. First observe that any reasonable adversary would not kill more that $P_i/\log P_i$ processors in round r_i, since it would not cause more work than $O(N + P_i\log P_i/\log\log P_i)$ (which is achieved when $f_i \geq P_i/\log P_i$). Therefore, we consider $f_i \leq P_i/\log P_i$ for all rounds r_i. Hence, the work in every round r_i (per Theorem 1) is $O\left(N + P_i\log P_i/\log(P_i/f_i)\right) = O\left(N + P\log P/\log(P/f_i)\right)$.
Let $W(N, P, f)$ be this one-round upper bound. As $f = \sum f_i$, an upper bound on r-*Do-All*$^{\mathcal{O}_L}(N, P, f)$ can be given by maximizing $\sum_i W(N, P_i, f_i)$ over all such failure patterns. As $W(\cdot,\cdot,\cdot)$ is monotone in P, we may assume that $P_i = P$ for the purposes of the upper bound. We show that this maximum is attained at $f_1 = f_2 = \ldots = f_r$. For simplicity, treat f_i as a continuous parameter and consider the factor in the single round work expression (given above) that depends on f_i : $c/\log(\frac{P}{f_i})$, where c is the constant hidden by the $O(\cdot)$ notation.
The first derivative over f_i is

$$\frac{\partial}{\partial f_i}\left(c/\log\left(\frac{P}{f_i}\right)\right) = c/f_i(\log P - \log f_i)^2,$$

and its second derivative is

$$\frac{\partial^2}{\partial f_i^2}\left(c/\log\left(\frac{P}{f_i}\right)\right) = 2c/f_i^2(\log P - \log f_i)^3 - c/f_i^2(\log P - \log f_i)^2.$$

Observe that the second derivative is negative in the domain considered (assuming $P > 16$). 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 sum maximized when all f_is are equal, specifically when $f_i = f/r$.
As the above upper bound on the sum $\sum_i W(N, P_i, f_i)$ is valid over *all* f_i in this range, it holds in particular for the choices made by the adversary which must, of course, cause an integer number of faults in each round. Therefore,

$$W_2 = O\left(r \cdot \left(N + P\frac{\log P}{\log(\frac{Pr}{f})}\right)\right).$$

The result then follows from the definition of $\Lambda^r_{N,P,f}$ by combining the two cases. □

Theorem 4. *Given any algorithm that solves* r-*Do-All*$^{\mathcal{O}}(N, P, f)$ *for* $P \leq N$ *and under any oracle* $\mathcal{O}$*, there exists a crash adversary that causes work*

$$W = \Omega\left(r \cdot \left(N + P\log P/\Lambda^r_{N,P,f}\right)\right).$$

Proof. Consider two cases:

Case 1: $f > \frac{Pr}{\log P}$. In this case the adversary may crash $P/\log P$ 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 $\lceil r/2\rceil \cdot \Omega\left(N + P\log P/\log\log P\right) = \Omega\left(Nr + Pr\log P/\log\log P\right)$ work.

Case 2: $f \leq \frac{Pr}{\log P}$. In this case the adversary ideally would crash f/r processors in every round. It can do that in the case where r divides f. If this is not the case, then the adversary crashes $\lceil f/r \rceil$ processors in r_A rounds and $\lfloor f/r \rfloor$ 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 $\Omega\left(Nr + Pr \log_{Pr/f} P\right)$ lower bound for work. Note that we consider only the case where $r \leq f$; otherwise the work is trivially $\Omega(rN)$.

The result then follows from the definition of $\Lambda^r_{N,P,f}$ by combining the two cases. □

Remark 2 Consider the setting where a central server repeatedly allocates tasks to crash-prone processors. When a processor completes a task, it reports this to the server. If a server detects processor failures, it must re-allocate the tasks to other processors. Processor crashes might cause some tasks to be executed more than once. Our results obtained for *Do-All*$^{\mathcal{O}}(N, P, f)$ are relevant to the bounds on task execution redundancy in such a setting. When the server allocates N similar, independent and idempotent tasks to P synchronous, crash-prone processors, then, per Theorems 1 and 2, the total number of task executions is $\Theta\left(N + P + P \log(\min(N, P))/\Lambda_{N,P,f}\right)$, for $f < P$. □

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 $F \in \mathcal{F}$, 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 $\tau(F)$ in the presence of the failure pattern F, where $|F| \leq f$, then the message complexity M is:

$$M_{N,P,f} = \max_{F \in \mathcal{F},\ |F| \leq f} \left\{ \sum_{i \leq \tau(F)} M_i(F) \right\}.$$

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 [16] is available, however messages to/from faulty processors may be lost.

5.1.1 Description of the algorithm

We now give a brief description of the algorithm; additional details are given in Appendix A (but to avoid a complete restatement, 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 $\mathcal{O}_L$ in the algorithm in Fig. 1.

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

In the rest of this section we present the new analysis of work and message complexity of algorithm AN. Throughout we assume that the algorithm correctness is shown as in [4].

5.1.2 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 8. [4] *In any execution of algorithm* AN *with* $f < P \leq N$ *we have* $W_a = O\left(N + P \frac{\log P}{\log \log P}\right)$ *and* $W_u = O\left(W_a \log f\right)$.

We now give the new analysis of algorithm AN.

Lemma 9. *In any execution of algorithm* AN *with* $P \leq N$ *we have* $W_a = O\left(N + P \log_{\frac{P}{f}} P\right)$, *when* $f \leq \frac{P}{\log P}$.

Proof. Given a phase i of an execution of algorithm AN, we define P_i to be the number of live processors and U_i to be the number of undone tasks at the beginning of the phase ($P_0 = P$ and $U_0 = N$). Let $\alpha_1, \alpha_2, \ldots \alpha_\tau$, denote all the attended phases of this execution (α_τ is the last phase of the execution).

Observe that for all $\alpha_i,\ 1 \leq i \leq \tau - 1$ it holds that

- $U_{a_i} > U_{a_{i+1}}$, and
- $P_{a_i} \geq P_{a_{i+1}}$.

This follows from the construction of algorithm AN: Since phase α_i is attended, there is at least one coordinator, call it c, alive in phase α_i. c executes one task. Hence, at least one task is executed and consequently at least one task is removed from U_{a_i}. The number of processors can only decrease, since we do not allow restarts.

In [4], Sect. 3.2, it is shown that if at the beginning of phase a_i, the processors have consistent information on the number of surviving processors (P_{a_i}) and the number of remaining tasks (U_{a_i}), then the operational processors will have consistent information on $P_{a_{i+1}}$ and $U_{a_{i+1}}$ at the beginning of phase a_{i+1}. And since the processors have consistent information at a_0, that means that at the beginning of every attended phase, the surviving processors have consistent view of the system. Hence, the processors in attended phases can perform perfect load balancing, as in the case where the processors are assisted by the oracle $\mathcal{O}_L$, in the oracle model. Therefore, focusing only on the attended phases (and assuming that in the worst case no progress is made in unattended phases), we obtain the desired result by induction on the size of undone tasks U, as in the proof of Lemma 3. □

Theorem 5. *In any execution of algorithm* AN *with* $P \leq N$ *we have*

$$W = O\left(\log f(N + P \log P / \Lambda_{N,P,f})\right).$$

Proof. This follows from Lemmas 8 and 9, the fact that $W = W_a + W_u$ and the definition of $\Lambda_{N,P,f}$. □

5.1.3 New 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 10. [4] *In any execution of algorithm* AN *with* $P \leq N$ *we have* $M_a = O(W_a)$ *and* $M_u = O(fP)$.

Theorem 6. *In any execution of algorithm* AN *with* $P \leq N$ *we have*

$$M = O\left(N + P \log P / \Lambda_{N,P,f} + Pf\right).$$

Proof. It follows from Lemmas 8, 9 and 10, the fact that $M = M_a + M_u$ and the definition of $\Lambda_{N,P,f}$. □

5.2 Analysis of Message-Passing Iterative Do-All

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

Theorem 7. *The* r-*Do-All*(N, P, f) *problem can be solved on synchronous crash-prone message-passing processors with work*

$$W = O\left(r \cdot \log\left(\frac{f}{r}\right) \cdot \left(N + P \log P / \Lambda^r_{N,P,f}\right)\right)$$

and with message complexity

$$M = O\left(r \cdot \left(N + P \log P / \Lambda^r_{N,P,f}\right) + fP\right).$$

Proof. The iterative *Do-All* can be solved by running algorithm AN on r instances of size N in sequence. We call this algorithm AN*. To analyze the efficiency of AN* we use the same approach as in the proof of Theorem 3. In the current context we base our work complexity arguments on the result of Theorem 5, and we base our message complexity arguments on the result of Theorem 6. □

6 New bounds for the shared-memory model

In the shared-memory models the *Do-All* problem is better known as the *Write-All* problem: given an N-element shared array, set each element of the array to 1 using P processors, in the presence of any failure pattern in the failure model $\mathcal{F}$. Here the notion of a task is abstracted as writing to a specific shared-memory location. Since we abstract away the nature of the tasks as long as they can be performed in constant time, we continue referring to the *Do-All*(N, P, f) and the r-*Do-All* (N, P, f) problems so as not to introduce redundant terminology. Here we give a new refined analysis of the most work-efficient known algorithm for the shared-memory model, algorithm W [18]. 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. Its work for any pattern of crashes is $O(N + P \log N \log P / \log \log P)$ for $P \leq N$ [18]. Note that this bound is conservative, since it does not include f, the number of crashes.

6.1.1 Description of the algorithm

We now give a brief description of the algorithm; additional details are given in Appendix B (but to avoid a complete restatement, we refer the reader to [17]). 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 the oracle $\mathcal{O}_L$, 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 $O(\log N)$.

6.1.2 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 11. [17] *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 12. *For any failure pattern with $f \leq \frac{P}{\log P}$, the number of block-steps required by the P-processor algorithm* W *with U leaves in the progress tree is*

$$B = O\left(U + P\log_{\frac{P}{f}} P\right).$$

Proof. It is not difficult to see, that the processor block-steps are equivalent to the processor steps under the perfect load-balancing assumption. Hence, the proof is the same as the proof of Lemma 3. □

Theorem 8. *Algorithm* W *solves Do-All(N, P, f) for $P \leq N$, using work*

$$O\left(N + P\log N\log P/\Lambda_{N,P,f}\right).$$

Proof. We consider the following two cases:

Case 1: $P < \frac{N}{\log N}$. Here the number of leaves in the progress tree is $U = N/\log N$ and in the work phase W3 each processor performs $N/U = \log N$ tasks. The cost of a single block-step is $C_1 = O(\log N)$ since each of the four phases takes at most $\log N$ time. We consider two subcases:

$(1a)$ $f \leq \frac{P}{\log P}$. Per Lemma 12, the number of blocks-steps B_{1a} for this case is:

$$B_{1a} = O\left(U + P\frac{\log P}{\log\frac{P}{f}}\right) = O\left(\frac{N}{\log N} + P\frac{\log P}{\log\frac{P}{f}}\right).$$

Therefore,

$$W_{1a} = B_{1a}\cdot C_1 = O\left(\frac{N}{\log N} + P\frac{\log P}{\log\frac{P}{f}}\right)\cdot O(\log N)$$
$$= O\left(N + P\log N\frac{\log P}{\log\frac{P}{f}}\right).$$

$(1b)$ $f > \frac{P}{\log P}$. Per Lemma 11, the number of block-steps B_{1b} for this case is:

$$B_{1b} = O\Big(U + P\frac{\log P}{\log\log P}\Big) = O\Big(\frac{N}{\log N} + P\frac{\log P}{\log\log P}\Big).$$

Therefore,

$$W_{1b} = B_{1b}\cdot C_1 = O\Big(\frac{N}{\log N} + P\frac{\log P}{\log\log P}\Big)\cdot O(\log N)$$
$$= O\Big(N + P\log N\frac{\log P}{\log\log P}\Big).$$

These two subcases together with the definition of $\Lambda_{N,P,f}$ yield $W_1 = O(N + P\log N\log P/\Lambda_{N,P,f})$.

Case 2: $\frac{N}{\log N} \leq P \leq N$. Here the number of leaves in the progress tree is $U = P$ and in the work phase W3 each processor performs $\lceil N/P\rceil = O(\log N)$ tasks. Thus the cost of a single block-step is $C_2 = O(\log N)$. We again consider two subcases:

$(2a)$ $f \leq \frac{P}{\log P}$. Per Lemma 12, the number of block-steps B_{2a} for this case is:

$$B_{2a} = O\left(U + P\frac{\log P}{\log\frac{P}{f}}\right) = O\left(P + P\frac{\log P}{\log\frac{P}{f}}\right)$$
$$= O\left(P\frac{\log P}{\log\frac{P}{f}}\right).$$

Therefore,

$$W_{2a} = B_{2a}\cdot C_2 = O\left(P\frac{\log P}{\log\frac{P}{f}}\right)\cdot O(\log N)$$
$$= O\left(P\log N\frac{\log P}{\log\frac{P}{f}}\right).$$

$(2b)$ $f > \frac{P}{\log P}$. Per Lemma 11, the number of block-steps B_{2b} for this case is:

$$B_{2b} = O\left(P + P\frac{\log P}{\log\log P}\right) = O\left(P\frac{\log P}{\log\log P}\right).$$

Therefore,

$$W_{2b} = B_{2b}\cdot C_2 = O\left(P\frac{\log P}{\log\log P}\right)\cdot O(\log N)$$
$$= O\left(P\log N\frac{\log P}{\log\log P}\right).$$

These last two subcases and the definition of $\Lambda_{N,P,f}$ yield $W_2 = O(P\log N\log P/\Lambda_{N,P,f})$. Combining Case 1 and Case 2 we get that $W = O(N + P\log N\log P/\Lambda_{N,P,f})$, for $1 \leq P \leq N$. □

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 ($P \leq N$), using shared memory, with work*

$$W = O\left(r \cdot \left(N + P \log N \log P / \Lambda^r_{N,P,f}\right)\right).$$

Proof. The iterative *Do-All* can be solved by running algorithm W on r instances of size N in sequence. We call this algorithm W*. To analyze the efficiency of W* we use the same approach as in the proof of Theorem 3. In the current context we base our work complexity arguments on the result of Theorem 8. □

Now we state another main result in this paper.

Theorem 10. *Any synchronous N-processor, r-time shared-memory parallel algorithm (PRAM) can be simulated on P crash-prone synchronous processors ($P \leq N$) with work*

$$O\left(r \cdot \left(N + P \log N \log P / \Lambda^r_{N,P,f}\right)\right).$$

Proof. The complexity of simulating a single parallel step of N ideal processors on P crash-prone processors does not exceed the complexity of solving a single *Do-All*(N, P, f) instance [21,28]. The result then follows from Theorem 9. □

7 Conclusion

In this paper we give the first complete analysis of the *Do-All* problem under the perfect load-balancing assumption. We introduce and analyze 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. Another recent work that emphasizes failure-sensitive analysis for *Do-All* is [5].

Our results also yield insight about the bounds on task execution redundancy incurred when a central authority repeatedly allocates tasks to crash-prone processors (cf. [22]). In particular, when similar, independent and idempotent tasks need to be performed, and the central authority has access to synchronous processors prone to crashes, then the bounds on the total number of task performed (counting redundant task executions) are precisely the bounds we established in this paper.

Acknowledgements. The authors thank Vassos Hadzilacos and the anonymous *Distributed Computing* reviewers whose comments and suggestions helped us improve the presentation of the results.

References

1. R.J. Anderson, H. Woll. Algorithms for the certified Write-All problem. *SIAM Journal of Computing* **26**(5), 1277–1283 (1997)
2. Y. Aumann, M.O. Rabin. Clock construction in fully asynchronous parallel systems and PRAM simulation. In *Proceedings of the 33^{rd} IEEE Symposium on Foundations of Computer Science (FOCS 1992)*, pp. 147–156, 1992
3. J. Buss, P.C. Kanellakis, P. Ragde, A.A. Shvartsman. Parallel algorithms with processor failures and delays. *Journal of Algorithms* **20**(1), 45–86 (1996)
4. B. Chlebus, R. De Prisco, A.A. Shvartsman. Performing tasks on restartable message-passing processors. *Distributed Computing* **14**(1), 49–64 (2001)
5. A. Clementi, A. Monti, R. Silvestri. Optimal f-reliable protocols for the the Do-All problem on single-hop wireless networks. In *Proceedings of the 13^{th} International Symposium on Algorithms and Computation (ISAAC 2002)*, pp. 320–331, 2002
6. P. Dasgupta, Z. Kedem, M. Rabin. Parallel processing on networks of workstation: A fault-tolerant, high performance approach. In *Proceedings of the 15^{th} IEEE International Conference on Distributed Computer Systems (ICDCS 1995)*, pp. 467–474, 1995
7. R. De Prisco, A. Mayer, M. Yung. Time-optimal message-efficient work performance in the presence of faults. In *Proceedings of the 13^{th} ACM Symposium on Principles of Distributed Computing (PODC 1994)*, pp. 161–172, 1994
8. S. Dolev, R. Segala, A.A. Shvartsman. Dynamic load balancing with group communication. *Theoretical Computer Science*, to appear. (Also in *Proceedings of the 6^{th} International Colloquium on Structural Information and Communication Complexity (SIROCCO 1999)*, pp. 111–125, 1999)
9. C. Dwork, J. Halpern, O. Waarts. Performing work efficiently in the presence of faults. *SIAM Journal on Computing* **27**(5), 1457–1491 (1998). A preliminary version appears in the *Proceedings of the 11^{th} ACM Symposium on Principles of Distributed Computing (PODC 1992)*, pp. 91–102, 1992
10. S. Fortune, J. Wyllie. Parallelism in random access machines. In *Proc. of the 10^{th} ACM Symposium on Theory of Computing (STOC 1978)*, pp. 114–118, 1978
11. Z. Galil, A. Mayer, M. Yung. Resolving message complexity of byzantine agreement and beyond. In *Proceedings of the 36^{th} IEEE Symposium on Foundations of Computer Science (FOCS 1995)*, pp. 724–733, 1995
12. Ch. Georgiou, A. Russell, A.A. Shvartsman. The complexity of distributed cooperation in the presence of failures. In *Proceedings of the 4^{th} International Conference on Principles of Distributed Systems (OPODIS 2000)*, pp. 245–264, 2000
13. Ch. Georgiou, A. Russell, A.A. Shvartsman. The complexity of synchronous iterative Do-All with crashes. In *Distributed algorithms (DISC 2001)*, volume 2180 of *Lecture Notes in Computer Science*, pp. 151–165, 2001
14. Ch. Georgiou, A.A. Shvartsman. Cooperative computing with fragmentable and mergeable groups. *Journal of Discrete Algorithms*, in press. (Also in *Proceedings of the 7^{th} International Colloquium on Structural Information and Communication Complexity (SIROCCO 2000),* pp. 141–156, 2000)
15. 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), 75–81 (2001)
16. V. Hadzilacos, S. Toueg. Fault-tolerant broadcasts and related problems. In *Distributed Systems*, chapter 5, pp. 97–145. ACM Press/Addison-Wesley, 1993
17. P.C. Kanellakis, A.A. Shvartsman. *Fault-Tolerant Parallel Computation*. Kluwer Academic Publishers, 1997

18. P.C. Kanellakis, A.A. Shvartsman. Efficient parallel algorithms can be made robust. *Distributed Computing* **5**(4), 201–217 (1992). A preliminary version appears in the *Proceedings of the* 8^{th} *ACM Symposium on Principles of Distributed Computing (PODC 1989)*, pp. 211–222, 1989
19. Z.M. Kedem, K.V. Palem, M.O. Rabin, A. Raghunathan. Efficient program transformations for resilient parallel computation via randomization. In *Proceedings of the* 24^{th} *ACM Symposium on Theory of Computing (STOC 1992)*, pp. 306–318, 1992
20. Z.M. Kedem, K.V. Palem, A. Raghunathan, P. Spirakis. Combining tentative and definite executions for dependable parallel computing. In *Proceedings of the* 23^{rd} *ACM Symposium on Theory of Computing (STOC 1991)*, pp. 381–390, 1991
21. Z.M. Kedem, K.V. Palem, P. Spirakis. Efficient robust parallel computations. In *Proceedings of the* 22^{nd} *ACM Symposium on Theory of Computing (STOC 1990)*, pp. 138–148, 1990
22. E. Korpela, D. Werthimer, D. Anderson, J. Cobb, M. Lebofsky. SETI@home: Massively distributed computing for SETI. *Computing in Science and Engineering* **3**(1), 78–83 (2001)
23. G. Malewicz, A. Russell, A.A. Shvartsman. Distributed cooperation during the absence of communication. In *Distributed algorithms (DISC 2000)*, volume 1914 of *Lecture Notes in Computer Science*, pp. 119–133, 2000
24. C. Martel, A. Park, R. Subramonian. Work-optimal asynchronous algorithms for shared memory parallel computers. *SIAM Journal on Computing* **21**(6), 1070–1099 (1992)
25. C. Martel, R. Subramonian. On the complexity of certified Write-All algorithms. *Journal of Algorithms* **16**(3), 361–387 (1994)
26. C. Martel, R. Subramonian, A. Park. Asynchronous PRAMs are (almost) as good as synchronous PRAMs. In *Proceedings of the* 31^{st} *IEEE Symposium on Foundations of Computer Science (FOCS 1990)*, pp. 590–599, 1990
27. R.D. Schlichting, F.B. Schneider. Fail-stop processors: An approach to designing fault-tolerant computing systems. *ACM Transactions on Computing Systems* **1**(3), 222–238 (1983)
28. A.A. Shvartsman. Achieving optimal CRCW PRAM fault-tolerance. *Information Processing Letters* **39**(2), 59–66 (1991)

Chryssis Georgiou received a B.S. in Mathematics from the University of Cyprus (Nicosia, Cyprus) in 1998 and an M.S. in Computer Science and Engineering from the University of Connecticut (Storrs, CT) in 2002. Currently he is a Ph.D. candidate in the Department of Computer Science and Engineering at the University of Connecticut. His research interests focus on the design and complexity analysis of fault-tolerant distributed and parallel algorithms.

Alexander Russell was born in Philadelphia, PA in 1969. He received a B.A. in Mathematics and a B.A. in Computer Science from Cornell University in 1991. He attended the Massachusetts Institute of Technology from 1991 to 1996, earning an S.M. in Electrical Engineering and Computer Science in 1993 and a Ph.D. in Mathematics in 1996. He was a postdoctoral fellow at the Royal Institute of Technology in Stockholm, Sweden, during 1996 and 1997 and then held a two year joint postdoctoral appointment at the University of Texas at Austin and the University of California, Berkeley. He is currently an associate professor at the University of Connecticut, Storrs. His research interests include complexity theory, cryptography, pseudorandomness, quantum computing, combinatorics, and harmonic analysis.

Alexander Allister Shvartsman is on the faculty of Computer Science and Engineering at the University of Connecticut and he is a research associate at the Laboratory for Computer Science of the Massachusetts Institute of Technology. Previously he was a Member of Technical Staff at Bell Labs from 1981 to 1982, and later he had led the development of distributed systems in the areas of manufacturing automation, resource management and interactive multimedia at Digital Equipment Corp. from 1983 to 1994 and Logica, Inc. from 1994 to 1995. His primary research interests are in the principles and practices of dependable distributed computing. He graduated secondary school in 1972 in Chişinău, Moldova. He received a B.S. from Stevens Institute of Technology in 1979, an M.S. from Cornell University in 1981, and a Ph.D. from Brown University in 1992, all in Computer Science.

Appendix

A. Algorithm AN

The following information about algorithm AN is taken from [4]. More information can be obtained there. The algorithm proceeds in a *loop* which is repeated 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 (thus a phase consists of 9 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. We refer to these three step as the *receive* substage, the *compute* substage and the *send* substage.

Coordinators and workers. A processor can be a *coordinator* of a given phase. All processors (including coordinators) are *workers* in a given phase. Coordinators are responsible for recording progress, while workers perform tasks and report on that to the coordinators. In the first phase one processor acts as the coordinator. There may be multiple coordinators in subsequent phases. 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. Whenever at least one coordinator survives a given phase, the number of coordinators for the next phase is reduced to one.

If at least one processor acts as a coordinator during a phase and it completes the phase without failing, we say that the phase is *attended*, the phase is *unattended* otherwise.

Local views. Processors assume the role of coordinator based on their local knowledge. During the computation each processor w maintains a list $L_w = \langle q_1, q_2, ..., q_k \rangle$ of supposed live processors. We call such list a *local view*. The processors in L_w are partitioned into *layers* consisting of consecutive sublists of L_w: $L_w = \langle \Lambda^0, \Lambda^1, ..., \Lambda^j \rangle$. The number of processors in layer Λ^{i+1}, for $i = 0, 1, ..., j-1$, is the double of the number of processors in layer Λ^i. Layer Λ^j may contain less processors. When $\Lambda^0 = \langle q_1 \rangle$ the local view can be visualized as a binary tree rooted at processor q_1, where nodes are placed from left to right with respect to the linear order given by L_w. Thus, in a tree-like local view, layer Λ^0 consists of processor q_1, layer Λ^i consists of 2^i consecutive processors starting at

processor q_{2^i} and ending at processor $q_{2^{i+1}-1}$, with the exception of the very last layer that may contain a smaller number of processors. Processors in a local view do not necessarily appear in the order of processor identifiers.

The local view is used to implement the martingale principle of appointing coordinators as follows. Let $L_{\ell,w} = \langle \Lambda^0, \Lambda^1, ..., \Lambda^j \rangle$ be the local view of worker w at the beginning of phase ℓ. Processor w expects processors in layer Λ^0 to coordinate phase ℓ; if no processor in layer Λ^0 completes phase ℓ, then processor w expects processors in layer Λ^1 to coordinate phase $\ell + 1$; in general processor w expects processors in layer Λ^i to coordinate phase $\ell + i$ if processors in all previous layers Λ^k, $\ell \leq k < \ell + i$, did not complete phase $\ell + k$. The local view is updated at the end of each phase.

Phase structure and task allocation. The structure of a phase of the algorithm is as follows. Each processor w keeps its local information about the set of tasks already performed, denoted D_w, and the set of live processors, denoted P_w, as known by processor w. Set D_w is always an underestimate of the set of tasks actually done and P_w is always an overestimate of the set of processors that are "available" from the start of the phase. We denote by U_w the set of *unaccounted* tasks, i.e., whose done status is unknown to w. Sets U_w and D_w are related by $U_w = \mathcal{T} \setminus D_w$, where $\mathcal{T}$ is the set of all the tasks. Given a phase ℓ we use $P_{\ell,w}$, $U_{\ell,w}$ and $D_{\ell,w}$ to denote the values of the corresponding sets at the beginning of phase ℓ.

Computation starts with phase 0 and any processor q has all processors in $L_{0,q}$ and has $D_{0,q}$ empty. At the beginning of phase ℓ each worker (that is, each processor) w performs one task according to its local view $L_{\ell,w}$ and its knowledge of the set $U_{\ell,w}$ of unaccounted tasks, using the following *load-balancing rule*. Worker w executes the task whose rank is $i \bmod |U_{\ell,w}|$ in the set $U_{\ell,w}$ of unaccounted tasks, where i is the rank of processor w in the local view $L_{\ell,w}$. Then the worker reports the execution of the task to all the processors that, according to the worker's local view, are supposed to be coordinators of phase ℓ. For simplicity we assume that a processor sends a message to itself when it is both worker and coordinator.

Any processor c that, according to its local view, is supposed to be coordinator, gathers reports from the workers, updates its information about $P_{\ell,c}$ and $U_{\ell,c}$ and broadcasts this new information causing the local views to be reorganized. In [4] it is shown that at the beginning of any phase ℓ all live processors have the same local view L_ℓ and the same set U_ℓ of unaccounted tasks and that accounted tasks have been actually executed. A new phase starts if U_ℓ is not empty. The detailed description of a phase is given in Fig. 2.

Local view update rule. In phase 0 the local view $L_{0,w}$ of any processor w is a tree-like view containing all P processors ordered by their PIDs. Let $L_{\ell,w} = \langle \Lambda^0, \Lambda^1, ..., \Lambda^j \rangle$ be the local view of processor w for phase ℓ. We distinguish two possible cases.

Case 1. Phase ℓ is unattended. Then the local view of processor w for phase $\ell + 1$ is $L_{\ell+1,w} = \langle \Lambda^1, ..., \Lambda^j \rangle$.

Case 2. Phase ℓ is attended. Then processor w receives `summary` messages from some coordinator in Λ^0. Processor w computes its set P_w as described in stage 3 (we will see that all processors compute the same set P_w). The local view $L_{\ell+1,w}$ of w for phase $\ell + 1$ is a tree-like local view containing the processors in P_w ordered by their PIDs.

Stage 1. The receive substage is not used. In the compute substage, any processor w performs a specific task z according to the load-balancing rule. In the send substage processor w sends a `report`(z) to any coordinator, that is, to any processor in the first layer of the local view $L_{\ell,w}$.

Stage 2. In the receive substage the coordinators gather `report` messages. For any coordinator c, let $z_c^1, \ldots, z_c^{k_c}$ be the set of TIDs received. In the compute substage c sets $D_c \leftarrow D_c \cup \bigcup_{i=1}^{k_c} \{z_c^i\}$, and P_c to the set of processors from which c received `report` messages. In the send substage, coordinator c multicasts the message `summary`(D_c, P_c) to processors in P_c.

Stage 3. During the receive substage `summary` messages are received by live processors. For any processor w, let $(D_w^1, P_w^1), \ldots, (D_w^{k_w}, P_w^{k_w})$ be the sets received in `summary` messages. In the compute substage w sets $D_w \leftarrow D_w^i$ and $P_w \leftarrow P_w^i$ for an arbitrary $i \in \{1, \ldots, k_w\}$ and updates its local view L_w as described below. The send substage is not used.

Fig. 2. Phase ℓ of algorithm AN

Finally, given an execution of algorithm AN, we enumerate its phases as follows. We denote the attended phases of the execution by $\alpha_1, \alpha_2, \ldots$, etc. We denote by π_i the sequence of unattended phases between the attended phases α_i and α_{i+1}. We refer to π_i as the i^{th} (unattended) period; an unattended period can be empty. Hence the computation proceeds as follows: unattended period π_0, attended phase α_1, unattended period π_1, attended phase α_2, and so on.

B. Algorithm W

The following information about algorithm W is taken from [17, Sect. 3.3.1]; more information can be obtained there. Algorithm W consists of the 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 work remaining and controls the parallel loop. The high level structure of the algorithm is given in Fig. 3.

The entire algorithm is moderately involved, but fairly modular. Phases W1 and W4 involve bottom-up traversal of two different trees and phase W2 involves a top-down traversal of these trees.

Input: Shared array $x[1..N]$; $x[i] = 0$ for $1 \leq i \leq N$.

Output: Shared array $x[1..N]$; $x[i] = 1$ for $1 \leq i \leq N$.

Data structures: The algorithm uses full binary trees to (1) enumerate surviving processors, (2) allocate processors, (3) perform work (modelled by $x[i] := 1$), and (4) measure progress. There are four full binary trees, each of size $2N - 1$, stored as linear arrays in shared memory. The trees are $c[1..2N - 1]$ (for processor counting and allocation),

```
forall processors PID=1..P parbegin
  Phase W3: Perform work on the input data
            at leaves based on PID
  Phase W4: Traverse the d tree bottom up
            to measure progress
  while the root of the d tree is not N do
    Phase W1: Traverse trees c and cs bottom-up
              to enumerate processors
    Phase W2: Traverse trees d, a, c top-down
              to reschedule work
    Phase W3: Perform rescheduled work on the
              input data
    Phase W4: Traverse the d tree bottom up
              to measure progress
  od
parend.
```

Fig. 3. A high level view of algorithm W

$cs[1..2N-1]$ (for counting step numbers), $d[1..2N-1]$ (for progress counting) and $a[1..2N-1]$ (for top-down auxiliary accounting). They are initially 0. The root of each tree is stored at location 1 of the corresponding array. An interior node at location i has its left child stored at location $2i$ and its right child stored at location $2i+1$.

The input is in a shared array $x[1..N]$, where the N elements of this array are associated with the leaves of the trees d and a. Element $x[i]$ is associated with $d[i+N-1]$ and $a[i+N-1]$, where $1 \leq i \leq N$. Similarly processors are initially associated with the leaves of the tree c, such that processor PID is associated with $c[\text{PID}+N-1]$.

Each processor uses some constant amount of local memory. For example, this local memory may be used to perform some simple arithmetic computations. Important local variables are PID, containing the initial processor identifier, and pn, containing a dynamically changing processor number.

Control flow. Due to the omniscience of the adversary, we employ an oblivious iterative approach in the sense that the pool of the available processors is treated uniformly and is assigned evenly to the tasks that need to be done. The basic idea of the *loop* is: (a) For *failure detection* use bottom-up, fast parallel summation to estimate the surviving processors and to estimate the progress they have made. (b) For *load rescheduling* use a top down, divide-and-conquer strategy based on the estimate of progress made.

The algorithm consists of the parallel *loop* given in Fig. 3. This loop is performed synchronously by all processors that have not stopped. It consists of four phases of steps, and the first time only part of it is executed (phases W3 and W4). Of course, processors can crash at any time during the algorithm. We now give a high level description of the phases.

Phase W1: *Failure detection via processor enumeration*. The processors traverse bottom-up a full binary tree used for processor counting starting with the leaves associated with processor identifiers (PIDs) and finishing at the root. Processor counting is implemented as a version of the standard logarithmic time parallel summation algorithm.

Phase W2: *Failure recovery via processor reallocation*. The processors use the full binary tree that represents the progress of the algorithm, and traverse it starting with the root and finishing at the leaves associated with the unfinished work. The processors are allocated in a divide-and-conquer fashion according to the hierarchy of the progress tree.

Phase W3: *The work phase*. The processors now perform work they find at the leaves they reached in phase W2. (This work is a simple assignment in the case of *Write-All*, and other specific tasks when the algorithm is used for robust simulations or transformations.)

Phase W4: *The progress measurement phase*. The processors begin at the leaves of the progress tree where they ended phase W3 and traverse bottom-up it to the root to estimate the progress of the algorithm. The counting of the number of leaves where the work of phase W3 was successfully done is implemented as a version of the standard parallel summation algorithm.