Randomization Helps to Perform Tasks on Processors Prone to Failures

Bogdan S. Chlebus* Dariusz R. Kowalski*

Abstract

The problem of performing t tasks in a distributed system of p processors is studied. The tasks are assumed to be independent, similar (each takes one step to be completed), and idempotent (can be performed many times and concurrently). The processors communicate by passing messages and each of them may fail. This problem is usually called DO-ALL, it was introduced by Dwork, Halpern and Waarts.

The distributed setting considered in this paper is as follows: The system is synchronous, the processors fail by stopping, reliable multicast is available. The occurrence of faults is modeled by an adversary who has to choose at least $c \cdot p$ processors prior to the start of the computation, for a fixed constant $0 < c < 1$, and then may fail any of the remaining processors at any time.

The main result is showing that there is a sharp difference between the expected performance of randomized algorithms versus the worst-case deterministic performance of algorithms solving the DO-ALL problem in such a setting.

Performance is measured in terms of work and communication of algorithms. Work is the total number of steps performed by all the processors while they are operational, including idling. Communication is the total number of point-to-point messages exchanged. Let effort be the sum of work and communication. A randomized algorithm is developed which has the expected effort $\mathcal{O}(t + p \cdot (1 + \log^* p - \log^*(p/t)))$, where $\log^*$ is the number of iterations of the log function required to go with the value of function down to 1. For deterministic algorithms and their worst-case behavior, we prove a lower bound $\Omega(t + p \cdot \log t / \log\log t)$ on work, and show that it is matched by the work performed by a simple algorithm.

Key words: distributed algorithm, message passing, stop failures, independent tasks, randomized algorithm, work, lower bound.

*Instytut Informatyki, Uniwersytet Warszawski, Banacha 2, 02-097 Warszawa, Poland. Email: {chlebus,darek}@mimuw.edu.pl. This research was supported by KBN contract 8 T11C 036 14.

1 Introduction

The problem of performing t tasks in a distributed system of p processors is considered. The tasks are assumed to be independent, similar, and idempotent. The processors communicate by passing messages and are prone to failures. This problem is usually called DO-ALL.

Review of prior work. The problem DO-ALL was introduced by Dwork, Halpern and Waarts [5], and subsequently studied in [3, 4, 6]. In these papers, if processors fail by stopping then the algorithms are required to perform all the tasks even if only one processor survives. Dwork, Halpern and Waarts [5] consider work defined as the total number of tasks performed counting multiplicities, and communication measure defined as the total number of messages sent. They develop three protocols solving the DO-ALL problem, all of them work optimal. They also consider the effort complexity defined as the sum of work and communication. One of their algorithms has effort complexity $\mathcal{O}(t + p\sqrt{p})$. De Prisco, Mayer and Yung [4] consider work defined as the available processor steps, that is, each processor contributes a unit to the measure of work for each step when it is operational. They present algorithm with work $\mathcal{O}(t + (f + 1)p)$ and communication $\mathcal{O}((f + 1)p)$, where f is the number of faults. In their algorithm, one processor is designated as an active coordinator which allocates tasks and receives reports of their completion. The coordinator may change over time. In [4] a lower bound $\Omega(t+(f+1)p)$ on work is proved for any algorithm that uses the checkpointing strategy. Another algorithm is developed by Galil, Mayer and Yung [6]. Paper [6] presents a solution of the Byzantine Agreement in the presence of crash failures using the optimal linear number of messages. The algorithmic technique used in the solution allows to improve the message complexity of the algorithm in [4] to $\mathcal{O}(fp^{\epsilon} + \min\{f + 1, \log p\}p)$, for any positive ϵ, while achieving the available processor steps $\mathcal{O}(t + (f + 1)p)$. Chlebus, De Prisco and Shvartsman [3] present two algorithms based on an aggressive coordination paradigm by which multiple coordinators may be active as the result of failures. One algorithm is tolerant of fail-stop failures, it has available processor steps $\mathcal{O}((t+p\log p/\log\log p)\log f)$ and communication $\mathcal{O}(t + p\log p/\log\log p + fp)$. The other algorithm is the only one known in the literature that tolerates restarts. Its available processor steps complexity is $\mathcal{O}((t + p\log p + f) \cdot \min\{\log p, \log f\})$, and its message complexity is $\mathcal{O}(t + p\log p + fp)$.

Summary of contributions. We present a randomized algorithm solving the DO-ALL problem and show that its expected performance is much better then worst-case behavior of any deterministic algorithm. The system is assumed to be synchronous, and a reliable multicast is available. The processors fail by crashing. The failures are controlled by an adversary who has to choose at least $c\cdot p$ processors prior to the start of the computation, where $0 < c < 1$ is a fixed constant, and then may fail any of the remaining processors at arbitrary times. The performance of algorithms is measured by work, which is the available processor steps, and communication, which is the total number of point-to-point messages. Define effort to be work plus communication. Let $\log^*$ be the number of iterations of the log function required to go with the value of function down to 1; formally $\log^{(1)} x = \log x$, $\log^{(k+1)} x = \log(\log^{(k)} x)$, and $\log^* x = \min_k[\log^{(k)} x \leq 1]$. Our randomized algorithm has the expected effort $\mathcal{O}(t + p(1 + \log^* p - \log^* \frac{p}{t}))$. The efficiency of this randomized algorithms is compared with the worst-case behavior of deterministic algorithms. A lower bound $\Omega(t + p\log t/\log\log t)$ is proved for the amount of work performed in the worst case by any deterministic algorithm. For instance, if $t = \Theta(p)$ then the randomized algorithm has the expected work $\mathcal{O}(p \cdot \log^* p)$, but any deterministic algorithm requires work $\Omega(p\log p/\log\log p)$ in the worst case. Given a deterministic algorithm, the adversary knows the future behavior of processors and may compare all the possible scenarios of failures; in particular if the adversary wants to fail a specific processor at a specific step to prohibit it from performing a certain task at this step, then also all the other processors (if any) assigned for the same task need to be failed simultaneously. Randomization helps because the adversary has no way to guess the pattern of assignments among tasks and processors if it is created on-line in a random fashion. We also show that the lower bound on work performed by any deterministic algorithms is matched by the worst-case work performance of a simple algorithm, which has the communication complexity $\mathcal{O}(t + p^2\log t/\log\log t)$.

Other related work. In shared-memory models there is a natural problem corresponding to DO-ALL. Given an array initialized with zeroes in the shared memory, a task is to change the value stored at a location of the array. The problem of performing all these tasks is called WRITE-ALL, it was introduced by Kanellakis and Shvartsman [9, 10]. Efficient solutions to this problem can be used iteratively to convert arbitrary shared-memory computations into robust ones resilient to processor failures. This was the subject of papers by Kedem, Palem and Spirakis [13], Kedem, Palem, Rabin, and Raghunathan [11], Martel, Park, and Subramonian [15], and Anderson and Woll [1]. A logarithmic lower bound on time to solve WRITE-ALL deterministically was derived by Kedem, Palem, Raghunathan and Spirakis [12], and on expected time of randomized executions by Martel and Subramonian [14]. Kanellakis and Shvartsman [9] proved a lower bound $\Omega(n\log n/\log\log n)$ on available processor steps in the model with memory snapshots, and developed an algorithm with a matching performance, where n denotes both the number of processors and the size of the array. Buss, Kanellakis, Ragde and Shvartsman [2] proved a lower bound $\Omega(n\log n)$ on available processor steps for any deterministic executions controlled by an adversary that can cause both failures and restarts, and developed an algorithm of a matching performance in the model with memory snapshots. For deterministic computations, the most efficient algorithm performs work $\mathcal{O}(n\log^2 n/\log\log n)$, and if restarts are allowed then the best upper bound is $\mathcal{O}(n^{1+\epsilon})$ for arbitrary positive ϵ; the best lower bound is $\Omega(n\log n)$ in both cases. For randomized computations, expected work $\mathcal{O}(n\log n)$ can be achieved even with restarts.

Contents of the paper. Section 2 presents details of the model and algorithmic paradigms. In Section 3 we present a randomized algorithm RA, prove its correctness and analyze its expected performance. Section 4 contains a proof of a lower bound on work performed in the worst case by any deterministic algorithm, and algorithm DA of a matching work performance. Section 5 contains final remarks.

2 Model and algorithmic preliminaries

Distributed setting. There are p synchronous processors which communicate by passing messages. There is no upper bound on the size of messages. A *stage* of computation consists of three clock cycles called *substages*: *receive* (receiving messages), *compute* (performing local computations), and *send* (multicasting a message). A message sent at a clock cycle is received during the next clock cycle. A processor may in one clock cycle multicast a message to any selected subset of processors, or receive all the incoming messages. A local computation may include performing a single task. The broadcast or multicast operations are reliable (cf. [8]), in the sense that if a processor $\mathcal{A}$ is attempting to perform such an operation and fails in this clock cycle then either no processor receives a message sent by $\mathcal{A}$ in this step or the messages sent by $\mathcal{A}$ in this clock cycle are successfully delivered to all the operational recipients (in the next step).

Tasks. There are t tasks to be performed, all the processors know the number t and each processor may at any clock cycle perform any specific task. The tasks are assumed to be independent, what means that any relative order of performing them is allowed. The tasks are similar, this means that each of them takes one step to be completed. Tasks are also idempotent, this means that they can be performed many times and concurrently.

Model of failure. The processors are prone to stop-failures. If a processor crashes then it stops performing any activity, and never restarts. The failures are controlled by an adversary. There is a fixed constant $0 < c < 1$, such that the adversary has to choose at least $c \cdot p$ processors prior to the start of the computation, and then may fail any of the remaining ones at any time. The algorithms presented in this paper do not resort to this constant c. Such an adversary has some features of the off-line adversary (choosing a subset of processors prone to failures in advance) and some of the adaptive adversary (failing the processors prone to failures on-line).

Performance measures. Two independent complexity measures are considered: work and communication. *Work* means the available processor steps, as defined by Kanellakis and Shvartsman [9, 10]. It counts all the steps performed by operational processors, including busy waiting; in other words, any operational processor contributes one unit to work for each clock cycle when it is not faulty. *Communication* is measured as the total number of point-to-point messages sent over the network, in particular a processor broadcasting a message to all the other processors contributes $p-1$ to the communication complexity. *Effort* is the sum of work and communication.

Coordinators and workers. In the course of an algorithm, the processors may be categorized into *coordinators* and the remaining ones, referred to as *workers*. A coordinator maintains this status till the end of computation, unless it fails, but a processor with the current status of a worker may later become a coordinator. The job of a coordinator is to repeatedly collect incoming messages about the progress of work, combine them, and send back to all the processors which are not known to have failed yet. Sending out such a message by a coordinator also serves the purpose of confirming that this coordinator is still alive. A message sent to every coordinator is called a *report*, and a message of a coordinator sent to every processor is a *summary*.

Phases. The computation proceeds in *phases* comprised of four consecutive stages. A phase may be performed in one of two modes: *work* or *election*. In the work mode a phase consists of the following four stages: (1) performing a task and sending reports, (2) receiving reports by the coordinators, combining them into summaries and sending the summaries, (3) receiving summaries, updating the local knowledge, and deciding on the mode of the next phase, (4) a pause. Each processor, including the coordinators, sends a message about the tasks done to every coordinator that either proclaimed itself a coordinator or confirmed its existence in the previous phase. Each processor maintains a list of tasks of which it does not know as being performed yet, and a list of processors of which it does not know as having failed yet. Each time a processor sends a report to the coordinators or a coordinator sends a summary to all the processors then these two whole lists are carried in the message. If a processor is sure that all the tasks have been completed, then it terminates. If no coordinator is heard during a phase then the remaining operational processors start an election process to select new coordinators, this is done by switching to the election mode. During an election phase, a worker may decide to be a coordinator, then it proclaims this by broadcasting a suitable message to all the processors it expects to be operational.

Local knowledge. Each processor $\mathcal{A}$ maintains three lists: list of outstanding tasks $\texttt{Tasks}_\mathcal{A}$, list of operational processors $\texttt{Processors}_\mathcal{A}$, list of coordinators $\texttt{Coordinators}_\mathcal{A}$. Lists $\texttt{Tasks}_\mathcal{A}$ and $\texttt{Processors}_\mathcal{A}$ are overestimates of the true situation because if a processor fails then it takes a number of steps for this information to reach every other processor, similarly if a task is performed. At the start of computation, each processor $\mathcal{A}$ obtains a list of all the tasks and a list of all the processors, sorts the lists and initializes $\texttt{Tasks}_\mathcal{A}$ with the list of all the tasks and $\texttt{Processors}_\mathcal{A}$ with the list of all the processors. Then $\mathcal{A}$ sets $\texttt{Coordinators}_\mathcal{A}$ to be empty. Message $M_\mathcal{A}$ that processor $\mathcal{A}$ sends includes the following data: the identification of $\mathcal{A}$, the coordinator/worker status of $\mathcal{A}$, and lists $\texttt{Tasks}_\mathcal{A}$ and $\texttt{Processors}_\mathcal{A}$.

3 Randomized algorithm

3.1 Algorithm RA

Each processor $\mathcal{A}$ maintains a number of local variables. One of them is the integer variable $\texttt{level}_\mathcal{A}$, initialized to 1. Is is *increased* by replacing the current value x by $2\cdot x$. Next is the integer variable $\texttt{coordination\_priority}_\mathcal{A}$, it is initialized to a random integer value in the range $[1..p]$, and never changed; the values of these variables of distinct processors are independent of each other. If the inequality $\texttt{level}_\mathcal{A} \geq \texttt{coordination\_priority}_\mathcal{A}$ holds for the first time during a certain phase then processor $\mathcal{A}$ is said to *qualify for a coordinator*, after that the processor becomes a coordinator. The summary sent by a coordinator in the first phase when it qualifies for a coordinator is called the *proclamation* of the coordinator. Another variable is the Boolean one mode which takes one of two values *work* and *election*. The computation starts from the election mode.

After initialization of the variables and lists, all the processors repeat four-stage phases in a loop until all of them terminate. Phases are either work or election, according to the mode. The following are detailed descriptions of phases.

Work phase of processor $\mathcal{A}$ of algorithm RA

STAGE 1: Receive substage not used. In compute substage perform a task: if the size of list $\texttt{Tasks}_{\mathcal{A}}$ is at least as large as the size of list $\texttt{Processors}_{\mathcal{A}}$ then perform task T such that the rank of T in $\texttt{Tasks}_{\mathcal{A}}$ is equal to rank of $\mathcal{A}$ in $\texttt{Processors}_{\mathcal{A}}$, otherwise select a random task from $\texttt{Tasks}_{\mathcal{A}}$ and perform it. Remove T from $\texttt{Tasks}_{\mathcal{A}}$. In send substage send report $M_{\mathcal{A}}$ to every processor on list $\texttt{Coordinators}_{\mathcal{A}}$.

STAGE 2: Let $\mathcal{A}$ be a coordinator, otherwise pause in this stage. In receive substage receive all the reports. In compute substage update two lists. On list $\texttt{Tasks}_{\mathcal{A}}$: remove any item T such that it is not included in some list $\texttt{Tasks}_{\mathcal{B}}$ contained in message $M_{\mathcal{B}}$ just received. On list $\texttt{Processors}_{\mathcal{A}}$: remove any item $\mathcal{B}$ such that no report of processor $\mathcal{B}$ was received. In send substage send the summary $M_{\mathcal{A}}$ to all the processors on list $\texttt{Processors}_{\mathcal{A}}$.

STAGE 3: In receive substage receive all the summaries. In compute substage update lists $\texttt{Tasks}_{\mathcal{A}}$ and $\texttt{Processors}_{\mathcal{A}}$ similarly as in Stage 2. Update list $\texttt{Coordinators}_{\mathcal{A}}$: for any item $\mathcal{B}$ on this list, if no summary received from it in Stage 2 then remove $\mathcal{B}$ from both lists $\texttt{Coordinators}_{\mathcal{A}}$ and $\texttt{Processors}_{\mathcal{A}}$. If list $\texttt{Coordinators}_{\mathcal{A}}$ is empty then switch the mode to election. If list $\texttt{Tasks}_{\mathcal{A}}$ is empty then terminate. Send substage not used.

STAGE 4: Pause for the whole stage.

Election phase of processor $\mathcal{A}$ of algorithm RA

STAGE 1: Receive substage not used. In compute substage increase $\texttt{level}_{\mathcal{A}}$. Then check if the test to qualify for a coordinator is passed. If this is the case then send proclamation $M_{\mathcal{A}}$ in send substage to every processor on list $\texttt{Processors}_{\mathcal{A}}$.

STAGE 2: In receive substage receive all the proclamations. In compute substage, for each proclamation $M_{\mathcal{B}}$ received add $\mathcal{B}$ to list $\texttt{Coordinators}_{\mathcal{A}}$. If this list is nonempty then in send substage send report $M_{\mathcal{A}}$ to every coordinator on it.

STAGE 3: Let $\mathcal{A}$ be a new coordinator, otherwise pause in this stage. Receive reports in receive substage. In compute substage: update lists $\texttt{Tasks}_{\mathcal{A}}$ and $\texttt{Processors}_{\mathcal{A}}$. In send substage send summary $M_{\mathcal{A}}$ to all the processors on list $\texttt{Processors}_{\mathcal{A}}$.

STAGE 4: In receive substage receive all the summaries. If at least one summary received then in compute substage update lists $\texttt{Tasks}_{\mathcal{A}}$, $\texttt{Processors}_{\mathcal{A}}$ and $\texttt{Coordinators}_{\mathcal{A}}$. If list $\texttt{Coordinators}_{\mathcal{A}}$ nonempty then switch the mode to work. Send substage not used.

3.2 Correctness of RA

A phase is *productive* if it ends in a work mode. Let *local view* of processor $\mathcal{A}$ consists of the values of variables $\texttt{mode}_{\mathcal{A}}$ and $\texttt{level}_{\mathcal{A}}$, and lists $\texttt{Tasks}_{\mathcal{A}}$, $\texttt{Processors}_{\mathcal{A}}$ and $\texttt{Coordinators}_{\mathcal{A}}$.

Lemma 1 *In eny execution of* RA*, the local view of any two operational processors is the same at the end of a productive phase.*

Proof: Induction on the number of phases. A phase is productive if at least one coordinator sent its summaries. They were received by all the processors by the inductive assumption and the reliability of multicast. The summaries were based on the same set of reports, by the same argument, hence updates of local lists of tasks, processors and coordinators produced the same result. The value of variable $\texttt{mode}_{\mathcal{A}}$ depends on list $\texttt{Coordinators}_{\mathcal{A}}$. It follows that all the processors simultaneously perform either work or election phases. Variable `level` is increased simultaneously by all the processors during election phases. □

Lemma 2 *Algorithm* RA *terminates after at most* $t + p + 2\log p$ *phases.*

Proof: The number of election phases is at most $\log(1 - c)p$ because the instances of variable `level` are simultaneously increased during each such a phase, by Lemma 1. Each work phase that is not productive precedes an election phase, hence the number of such phases is not larger than the number of election phases. It is enough to count the productive phases. Observe that during such a phase each processor $\mathcal{A}$ either removes a task from $\texttt{Tasks}_{\mathcal{A}}$ or adds a processor to $\texttt{Coordinators}_{\mathcal{A}}$. Namely, if it is a work phase, then each processor performs a task and removes it from its list of tasks, at the end of this phase all the processors have removed the task from their lists of tasks by Lemma 1. If this is an election phase then new coordinators are added to $\texttt{Coordinators}_{\mathcal{A}}$. It follows that the number of such phases is at most $t + p$. □

Theorem 1 *Algorithm* RA *terminates with all the tasks performed.*

Proof: By Lemma 2, algorithm RA terminates. Processor $\mathcal{A}$ terminates if list $\texttt{Tasks}_{\mathcal{A}}$ is empty. Processor $\mathcal{A}$ has removed a task from this list either if it performed it itself or received information in a summary that it had been performed. □

3.3 Analysis of RA

When processors select randomly tasks to be performed, the behavior of algorithm RA can be modeled by the following process of *clearing s urns with n balls*: at the start there are s empty urns, a step begins with a random placement of n balls in the given set of urns, then the urns containing at least one ball are removed; such steps are iterated until no urn remains.

Lemma 3 *The process of clearing s urns with n balls, for $n \geq s$, terminates within $\log^* n - \log^* \frac{n}{s} + \mathcal{O}(1)$ steps with the probability at least $1 - e^{-n^{1/5}}$, for sufficiently large n.*

Proof: If k balls are randomly placed in m urns then the expected number of empty urns is

$$m \cdot \Big(\frac{m-1}{m}\Big)^k = m \cdot \Big(1 - \frac{1}{m}\Big)^{m \cdot \frac{k}{m}} \leq m \cdot e^{-k/m} \ .$$

First consider the case $s = n$. Let X_i be the random variable equal to the number of empty urns after step i. We may assume that the balls in a step are placed one by one. Let $X_{i,k}$ be the number of empty urns after the k-th ball has been put in a random urn during step i.

Let $Y_{i,k} = \mathbf{E}[X_{i,k} \mid X_{i,1}, \ldots, X_{i,k-1}]$. Then the sequence $\langle Y_{i,k} \rangle_k$ is a martingale (cf. [7]). Adding one ball changes the number of empty urns by at most 1, hence $|Y_{i,k+1} - Y_{i,k}| \leq 1$. This allows to use the method of bounded differences (cf. [16]), and to estimate probabilities of deviations by the Azuma inequality, which takes the form

$$\mathbf{P}[X_i - \mathbf{E}\, X_i \geq x] \leq \exp(-x^2/2n) \ , \tag{1}$$

because $X_i = Y_{i,n}$. This inequality holds for any number of urns used in step i.

Let $e^{(k+1)} = e^{e^{(k)}}$, where $e^{(1)} = e$. We have $\mathbf{E}\, X_1 \leq n/e$. Then by (1) the inequality $\mathbf{P}[X_1 > \frac{n}{e} + \epsilon_1 n] \leq e^{-\delta_1 n}$ holds for each sufficiently small $\epsilon_1 > 0$ and some corresponding $\delta_1 > 0$. Let event $\mathcal{A}_1$ hold iff $X_1 < \frac{n}{e} + \epsilon_1 n$. Then

$$\mathbf{E}[X_2 \mid \mathcal{A}_1] \leq n(e^{-1} + \epsilon_1) \exp\Big(-\frac{n}{n(e^{-1} + \epsilon_1)}\Big) \to \frac{n}{e \cdot e^{(2)}}$$

with $\epsilon_1 \to 0$. Let event $\mathcal{A}_2$ hold iff $X_2 < n/e^{(2)}$. Then $\mathbf{P}[-\mathcal{A}_2 \mid \mathcal{A}_1] < e^{-\delta_2 n}$, for a certain $\delta_2 > 0$. The case of X_3 is similar to X_2, namely $\mathbf{E}[X_3 \mid \mathcal{A}_1 \cap \mathcal{A}_2] \leq n/(e^{(2)} e^{(3)})$. Let $\mathcal{A}_3$ hold iff $X_3 < n/e^{(3)}$. We can estimate the probability similarly as before: $\mathbf{P}[-\mathcal{A}_3 \mid \mathcal{A}_1 \cap \mathcal{A}_2] < \exp\big(-n/2(e^{(3)})^4\big)$. This leads to a generalization: define $\mathcal{A}_k$ to hold iff $X_k < n/e^{(k)}$ and $a_k = \mathbf{P}[-\mathcal{A}_k \mid \mathcal{A}_1 \cap \cdots \cap \mathcal{A}_{k-1}]$. Then the following inequalities hold:

$$\mathbf{E}[X_k \mid \mathcal{A}_1 \cap \cdots \cap \mathcal{A}_{k-1}] \leq \frac{n}{e^{(k-1)} e^{(k)}}$$

and

$$a_k < \exp\Big(\frac{-n}{2(e^{(k)})^4}\Big) \ .$$

Partition the n-process into two stages. Let the first stage terminate if the number of empty urns is less than $n/\log n$. If $k < \log^* x$ then $e^{(k)} \leq \log x$. We estimate the probability that the first stage terminates within $\log^* n - 1$ steps. Let $u = \log^* n$. The probability that all events $\mathcal{A}_i$ hold, for $3 \leq i < u = \log^* n$, is at least

$$\prod_{3 \leq i < u} (1 - a_i) = \exp\big(\ln \prod_{3 \leq i < u} (1 - a_i)\big) = \exp\big(\sum_{3 \leq i < u} \ln(1 - a_i)\big) \geq \exp(-2 \sum_{3 \leq i < u} a_i) \ ,$$

for $a_i < 1/2$. Also

$$a_i < \exp(-n/2(e^{(i)})^4) \leq \exp(-n/2 \log^4 n) \ ,$$

for $3 \leq i < u$. Hence $\sum_{3 \leq i < u} a_i < \log^* n \cdot \exp(-n/2\log^4 n)$. We need to estimate $\mathbf{P}\left[\bigcap_{1 \leq i < \log^* n} \mathcal{A}_i\right]$. This number is at least

$$\left(1 - e^{-\delta_1 n}\right)\left(1 - e^{-\delta_2 n}\right)\exp(-2 \cdot \log^* n \cdot e^{-n/2\log^4 n}) = 1 - \mathcal{O}\left(\frac{\log^* n}{e^{n/2\log^4 n}}\right) .$$

This is an estimation of the probability that the first stage is completed within $\log^* n - 1$ steps.

In stage two, the first step is of placing n balls into $n/\log n$ urns. Let Z be a random variable equal to the number of empty urns after this step, then

$$\mathbf{E}\, Z = \frac{n}{\log n} \cdot e^{-\log n} = \Theta(1/\log n) .$$

¿From the Azuma inequality:

$$\mathbf{P}\left[Z - \mathbf{E}\, Z \geq d\sqrt{n}\right] \leq \exp(-d^2/2) .$$

For $d = n^{1/4}$, with probability $\mathcal{O}(e^{-\sqrt{n}})$ there are $\mathcal{O}(n^{3/4})$ empty urns. Consider the next step of the second stage: the probability that a specific urn is empty is $\mathcal{O}((\frac{n^{3/4}-1}{n^{3/4}})^n) = \mathcal{O}(e^{-n^{1/4}})$. With probability $\mathcal{O}(n^{3/4} \cdot e^{-n^{1/4}})$ there is an empty urn after that step.

Thus the overall probability that after at most $1 + \log^* n$ steps there is no empty urn is at least

$$1 - \mathcal{O}\left(\log^* n \cdot \exp\left(-\frac{n}{2\log^4 n}\right)\right) - \mathcal{O}(\exp(-\sqrt{n})) - \mathcal{O}(n^{3/4} \cdot \exp(-n^{1/4})) \geq 1 - e^{-n^{1/5}} ,$$

for sufficiently large n.

Finally consider the case $s < n$. We need to count the number of steps needed to have at most $n/\log n$ empty urns. During each of these steps we estimate the conditional expected number of the remaining urns by number of the form $\frac{s}{e^{(k-1)}e^{(k)}}$, which is less than $n/\log n$ for $k = \log^* n - \log^* \frac{n}{s} + \mathcal{O}(1)$. The rest of the proof is the same as in the case $n = s$. □

Lemma 4 *The expected work of algorithm* RA *is* $\mathcal{O}(t + p \cdot (1 + \log^* p - \log^* \frac{p}{t}))$.

Proof: First consider the productive work phases. If during a phase the number of outstanding tasks is not smaller than the number of operational processors, then each task is performed by at most one processor, which is assigned to the task in a deterministic way. The total amount of work performed in this fashion is $\mathcal{O}(t)$. Now consider the phases when processors select tasks randomly. The number of operational processors is at least cp during each step. The process of diminishing the outstanding tasks is modeled by the process of clearing $\mathcal{O}(p)$ urns with $\Theta(p)$ balls. Hence the work performed is $\mathcal{O}(p(1 + \log^* p - \log^* \frac{p}{t}))$ by Lemma 3.

Next we estimate the number of election phases, which also gives a bound on the number of unproductive work phases. Let X be the minimum value of $\texttt{level}_{\mathcal{A}}$ for all the processors $\mathcal{A}$ that never fail. There are at least cp of such processors, and the following estimation holds:

$$\mathbf{E}\,X = \sum_{i=1}^{p} \mathbf{P}\,[X \geq i] \leq \sum_{i=1}^{p} \Big(1 - \frac{i-1}{p}\Big)^{cp} \leq \sum_{i=1}^{p} e^{-c(i-1)} = \mathcal{O}(1) \ .$$

Hence the expected number of election phases is $\mathcal{O}(1)$ and the expected work during such phases and the unproductive work phases is $\mathcal{O}(p)$. □

Lemma 5 *The expected number of messages sent while* RA *is executed is* $\mathcal{O}(t + p \cdot (1 + \log^* p - \log^* \frac{p}{t})$.

Proof: Let X be the minimum value of variable $\texttt{level}_{\mathcal{A}}$ for all the processors $\mathcal{A}$ that never fail. Let Y be the number of processors $\mathcal{B}$ such that the inequality $\texttt{level}_{\mathcal{B}} \leq 2^i$ holds, where i is the integer satisfying $2^{i-1} < X \leq 2^i$. Number $\mathbf{E}\,Y$ is the expected number of coordinators in the course of algorithm. The expected number of phases is $\mathcal{O}(1 + \frac{t}{p} + \log^* p - \log^* \frac{p}{t})$. The expected number of messages sent during a phase is $\mathcal{O}(p \cdot \mathbf{E}\,Y)$. We estimate $\mathbf{E}\,Y$ as follows:

$$\mathbf{E}\,(Y) = \sum_{i=1}^{p} \mathbf{E}\,(Y|X=i)\mathbf{P}\,(X=i) \leq \sum_{j=1}^{1+\log p} 2^j \mathbf{P}\,[2^{j-1} < X \leq 2^j]$$

$$\leq \sum_{j=1}^{1+\log p} 2^j \Big(1 - \frac{2^{j-1}}{p}\Big)^{cp} \leq \sum_{j=1}^{1+\log p} 2^j \exp\big(-c2^{j-1}\big) \leq \frac{1}{2} \sum_{k=1}^{\infty} k e^{-ck} = \mathcal{O}(1) \ .$$

Straightforward calculations complete the proof. □

Lemmas 4 and 5 imply:

Theorem 2 *The expected effort of algorithm* RA *is* $\mathcal{O}(t + p \cdot (1 + \log^* p - \log^* \frac{p}{t}))$. □

4 Deterministic computations

4.1 Lower bound for deterministic computation

We present specific strategies of the adversary which may force a large amount of work to be performed. Let p_i be the number of processor still operational after step i, where $p_0 = p$. Let t_i be the number of outstanding tasks after step i, where $t_0 = t$. Let $0 < a < 1$ be a parameter.

The strategy S_a is described by the following two rules:

Rule 1: In step $i+1$, if $t_i > 2p_i$ then let all the processors survive.

Rule 2: In step $i+1$, if $t_i \leq 2p_i$ then fail the smallest possible number of processors that can make the inequality $t_{i+1} \geq a \cdot t_i$ hold.

Notice that when the first processors are failed then the current number of outstanding tasks is at least p and at most $2p$. The adversary knows which processors are assigned to which tasks, and fails those processors that are assigned to the tasks with the smallest numbers of other processors also assigned to them. If Rule 2 starts to be applied then we never again apply Rule 1.

Lemma 6 *If $t \leq 2p$ then for any deterministic algorithm there is strategy S_a which forces time to be $\Omega(\frac{\log t}{\log \log t})$.*

Proof: Take strategy S_a for $a = \frac{1}{\log t}$. Only Rule 2 is applied. Then $\lceil at_{i-1} \rceil \geq t_i \geq at_{1-1}$ by Rule 2. The number of tasks performed in round i satisfies $t_{i-1} - t_i \leq (1-a)t_{i-1}$, and these tasks can be selected in such a way that the average number of tasks per processor is not greater than p_{i-1}/t_{i-1}. Hence

$$p_i \geq \frac{p_{i-1}}{t_{i-1}} \cdot (1-a)t_{i-1} = p_{i-1}(1-a) \ .$$

It follows by induction that $t_n \geq a^n \cdot t$ and $p_n \geq (1-a)^n \cdot p$. Substitute $n = \frac{\log t}{\log \log t}$. Then

$$t_n \geq \left(\frac{1}{\log t}\right)^{\frac{\log t}{\log \log t}} \cdot t = 1$$

and

$$p_n \geq \left(1 - \frac{1}{\log t}\right)^{\frac{\log t}{\log \log t}} \cdot p \geq 4^{-\frac{1}{\log \log t}} \cdot p \geq cp \ ,$$

for sufficiently large t. $\square$

Lemma 7 *If $t > 2p$ then for each deterministic algorithm there is a strategy S_a that forces work to be $\Omega(t + \frac{p \log p}{\log \log p})$.*

Proof: Apply strategy S_a for $a = \frac{1}{\log p}$. While Rule 1 applied, the amount of work is at least $t - 2p$ because each task performed by at most one processor. Consider the step when Rule 2 starts to be applied. Then the number t' of still outstanding tasks satisfies $p < t' \leq 2p$. The problem is reduced to the case $t \leq 2p$, and Lemma 6 applies. $\square$

Theorem 3 *Any deterministic algorithm requires work $\Omega(t + p \cdot \frac{\log t}{\log \log t})$ in the worst case.*

Proof: It follows from Lemmas 6 and 7 by asymptotic calculations. If $t \leq 2p$, then apply Lemma 6, because $t < \frac{p \log t}{\log \log t}$. If $\frac{t}{\log t} \leq 2p < t$, then use Lemma 7 and estimation

$$\frac{p \log p}{\log \log p} = \Omega\left(\frac{p \log(t/\log t)}{\log \log t}\right) = \Omega\left(\frac{p \log t}{\log \log t}\right) \ .$$

If $p < \frac{t}{\log t}$ then $\frac{p \log t}{\log \log t} < \frac{t}{\log t} \cdot \frac{\log t}{\log \log t} < t$. This combined with Lemma 7 gives

$$\Omega\left(t + \frac{p \log p}{\log \log p}\right) = \Omega(t) = \Omega\left(t + \frac{p \log t}{\log \log t}\right)$$

of work. $\square$

4.2 Deterministic algorithm

Algorithm DA is similar to RA. The main differences are as follows: (1) all the processors act as coordinators, and (2) tasks are performed in a load-balanced way, that is, processor $\mathcal{A}$ of rank i in list $\texttt{Processors}_{\mathcal{A}}$ performs the task of i-th rank mod the length of $\texttt{Tasks}_{\mathcal{A}}$. There are no election phases in DA, only work phases performed in a loop. Work phase consists of the first three stages of the work phase of RA, with $\texttt{Processors}_{\mathcal{A}}$ playing the role of $\texttt{Coordinators}_{\mathcal{A}}$, and the variables needed for elections not used. The correctness of DA is proved similarly as the correctness of RA.

Consider a sequence $\alpha = \langle a_i \rangle$, where $i = 1, \ldots, n$ and $0 < a_i \le 1$, for each $1 \le i \le n$. A pair of sequences $\langle p_i \rangle, \langle t_i \rangle$, where $i = 0, \ldots, n$, is an *α-scenario of length n* if the following conditions hold: $t_0 = t$, $p_0 = p$, $t_n \ge 1$, $p_i \le \lceil \frac{p_{i-1}}{t_{i-1}} \rceil t_{i-1}(1-a_i)$ and $\lceil t_{i-1} \cdot a_i \rceil \ge t_i \ge t_{i-1} \cdot a_i$, for $i = 1, \ldots, n$, and $p_i \ge cp$ for $i = 0, \ldots, n$. A *scenario* is an α-scenario for a certain suitable α.

Lemma 8 *Let $0 < a_1, \ldots, a_n < 1$. If $a^n = \prod_{i=1}^n a_i$ then $(1-a)^n \ge \prod_{i=1}^n (1-a_i)$.*

Proof: Take logarithms of both sides to obtain the equivalent inequality

$$n \cdot \ln(1-a) \ge \sum_{i=1}^{n} \ln(1-a_i) \ . \tag{2}$$

The left-hand side of (2) is $-n(a + \frac{a^2}{2} + \frac{a^3}{3} + \ldots)$ The right-hand side of (2) is $-(\sum_{i=1}^n a_i + \frac{1}{2}\sum_{i=1}^n a_i^2 + \frac{1}{3}\sum_{i=1}^n a_i^3 + \ldots)$. The inequalities $a^k \le \frac{1}{n}\sum_{i=1}^n a_i^k$ hold for all integer $k > 0$ because a^k is the geometric mean of $a_1^k, \ldots, a_n^k$, while $\frac{1}{n}\sum_{i=1}^n a_i^k$ is their arithmetic mean. Compare the expansions of both sides of (2) to obtain the inequality. $\Box$

Lemma 9 *Let $\langle p_i \rangle, \langle t_i \rangle$ be an α-scenario of length n. Then the following inequalities hold, for $1 \le i \le n$:*

$$p_i \le p \prod_{j=1}^{i} (1-a_j) + t(1-a_i)$$

and

$$\prod_{j=1}^{i} a_j \le t_i \le t \prod_{j=1}^{i} a_j + i \ .$$

Proof: We estimate p_i as follows:

$$\begin{aligned} p_i &\le \Big(\frac{p_{i-1}}{t_{i-1}} + 1\Big) t_{i-1}(1-a_i) = p_{i-1}(1-a_i) + t_{i-1}(1-a_i) \le \\ &[p_{i-2}(1-a_{i-1}) + t_{i-2}(1-a_{i-1})] \cdot (1-a_i) - t_{i-1}(1-a_i) \\ &\le p_{i-2}(1-a_{i-1})(1-a_i) + t_{i-2}(1-a_i) \ , \end{aligned}$$

and by induction obtain

$$p_i \le p \prod_{j=1}^{i} (1-a_j) + t(1-a_i) \ .$$

Numbers t_i satisfy the following inequalities:

$$t_i \le t_{i-1} \cdot a_i + 1 \le [t_{i-2} \cdot a_{i-1} + 1]a_i + 1 \le t_{i-2} \cdot a_{i-1} \cdot a_i + 2 \ ,$$

we obtain $t \prod_{j=1}^{i} a_j \le t_i \le t \prod_{j=1}^{i} a_j + i$ by induction. □

Lemma 10 *If $t \le cp$ then any α-scenario of length n satisfies $n \le \frac{2 \log t}{\log \log t}$.*

Proof: Assume to the contrary that $n = \frac{b \log t}{\log \log t}$ for a certain $b > 2$. By Lemma 8 we take a such that $a^n = \prod_{i=1}^{n} a_i$. Then we have also $(1-a)^n \ge \prod_{i=1}^{n}(1-a_i)$, and by Lemma 9 obtain

$$p_n \le p \prod_{i=1}^{n}(1-a_i) + t(1-a_n) \le p(1-a)^n + t(1-a_n) \ .$$

As number a we may take $h(\log t)^{-1/b}$, for a certain $h \ge 1$, because $\frac{1}{t} = \left(\frac{1}{\log^{1/b} t}\right)^{\frac{b \log t}{\log \log t}}$. We obtain $1 \le t_n \le i+1$, because $t \prod_{i=1}^{n} a_i + i \ge t_n \ge 1$ by Lemma 9. Then

$$(1-a)^n = \left(1 - \frac{h}{\log^{1/b} t}\right)^{\frac{b \log t}{\log \log t}} \le \exp\left(\frac{-hb \log^{1-1/b} t}{\log \log t}\right) \to 0$$

for sufficiently large t. Finally

$$p_n \le p(1-a)^n + t(1-a_n) \le p[(1-a)^n + c(1-a_n)] < cp \ ,$$

what contradicts the assumption $p_n \ge cp$. □

Theorem 4 *The algorithm* DA *performs work $\mathcal{O}(t + \frac{p \log t}{\log \log t})$ in the worst case.*

Proof: Let p_i and t_i be the numbers of operational processors and outstanding tasks after step i when algorithm DA executed. Then $\langle p_i \rangle, \langle t_i \rangle$ is a scenario. If $t \le cp$ then by Lemma 10 the work is $\mathcal{O}(\frac{p \log t}{\log \log t}) = \mathcal{O}(t + \frac{p \log t}{\log \log t})$. Suppose $t > cp$. Algorithm DA assigns at most $\lceil 1/c \rceil$ processors to each task during such a phase when there are more than cp outstanding tasks. The processors that never fail perform $\Omega(p)$ tasks during such a phase. Eventually the number of outstanding tasks becomes at most cp, while the work performed so far is $\mathcal{O}(t)$. By Lemma 10 the remaining work is $\mathcal{O}(\frac{p \log t}{\log \log t})$, because the function $\log x / \log \log x$ is increasing. □

The number of messages of algorithm DA is $\mathcal{O}(t + p^2 \cdot \frac{\log t}{\log \log t})$.

5 Discussion

We described the performance of algorithm RA in terms of the expected work and communication. If one would like to have bounds on work and communication holding with the probability polynomially close to 1, then such bounds for RA are $\mathcal{O}(t + p \log \log p)$ on work and $\mathcal{O}(t + p \log p)$ on communication.

Optimality of solving DO-ALL by a randomized algorithm in the setting of this paper is an open problem. We claim that the expected work of any such algorithm is $\Omega(p \log^* p)$ for $t = p$.

Acknowledgement. We thank Adam Malinowski for several discussions of randomized algorithms issues and for encouraging this direction of research.

References

[1] R.J. Anderson, and H. Woll, Wait-Free Parallel Algorithms for the Union-Find Problem, in *Proc. 23rd Symp. on Theory of Computing*, 1991, pp. 370–380.

[2] J. Buss, P.C. Kanellakis, P. Ragde, and A.A. Shvartsman, Parallel Algorithms with Processor Failures and Delays, *J. Algorithms*, 20 (1996) 45–86.

[3] B.S. Chlebus, R. De Prisco, and A.A. Shvartsman, Performing Tasks on Restartable Message-Passing Processors, in *Proc. 11th International Workshop on Distributed Algorithms*, 1997, LNCS 1320, pp. 96–110.

[4] R. De Prisco, A. Mayer, and M. Yung, Time-Optimal Message-Efficient Work Performance in the Presence of Faults, in *Proc. 13th Symp. on Principles of Distributed Computing*, 1994, pp. 161–172.

[5] C. Dwork, J. Halpern, O. Waarts, Performing Work Efficiently in the Presence of Faults, *SIAM J. on Computing*, 27 (1998) 1457–1491.

[6] Z. Galil, A. Mayer, and M. Yung, Resolving Message Complexity of Byzantine Agreement and Beyond, in *Proc. 36th Symp. on Foundations of Computer Science*, 1995, pp. 724–733.

[7] G. Grimmett, and D. Stirzaker, *"Probability and Random Processes,"* Oxford University Press, 1992.

[8] V. Hadzilacos and S. Toueg, Fault-Tolerant Broadcasts and Related Problems, in *"Distributed Systems,"* 2nd Ed., S. Mullender, Ed., Addison-Wesley and ACM Press, 1993.

[9] P.C. Kanellakis and A.A. Shvartsman, Efficient Parallel Algorithms Can Be Made Robust, *Distributed Computing*, 5 (1992) 201–217.

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

[11] Z.M. Kedem, K.V. Palem, M.O. Rabin, A. Raghunathan, Efficient Program Transformations for Resilient Parallel Computation via Randomization, in *Proc. 24th Symp. on Theory of Comp.*, 1992, pp. 306–318.

[12] Z.M. Kedem, K.V. Palem, A. Raghunathan, and P. Spirakis, Combining Tentative and Definite Executions for Dependable Parallel Computing, in *Proc. 23rd Symp. on Theory of Computing*, 1991, pp. 381–390.

[13] Z.M. Kedem, K.V. Palem, and P. Spirakis, Efficient Robust Parallel Computations, in *Proc. 22nd Symp. on Theory of Computing*, 1990, pp. 138–148.

[14] C. Martel, and R. Subramonian, On the Complexity of Certified Write-All Algorithms, *J. Algorithms*, 16 (1994) 361–387.

[15] C. Martel, A. Park, and R. Subramonian, Work-Optimal Asynchronous Algorithms for Shared Memory Parallel Computers, *SIAM J. Comput.*, 21 (1992) 1070–1099.

[16] Mc Diarmid, On the Method of Bounded Differences, in J. Siemon, ed., *"Surveys in Combinatorics,"* Cambridge University Press, 1989, pp. 148–188, London Math. Soc. Lecture Note Series 141.