Available online at www.sciencedirect.com
SCIENCE DIRECT®
Theoretical Computer Science 347 (2005) 130–166
Theoretical Computer Science
www.elsevier.com/locate/tcs

Efficient gossip and robust distributed computation☆

Chryssis Georgiou[a], Dariusz R. Kowalski[b], Alexander A. Shvartsman[c,*]

[a] *Department of Computer Science, University of Cyprus, CY-1678, Nicosia, Cyprus*
[b] *Instytut Informatyki, Uniwersytet Warszawski, 02-097 Warszawa, Poland*
[c] *Department of Computer Science and Engineering, University of Connecticut, Storrs, CT 06269, USA, and Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, MA 02139, USA*

Received 7 March 2004; received in revised form 4 March 2005; accepted 26 May 2005

Communicated by M. Mavronicolas

Abstract

This paper presents an efficient deterministic gossip algorithm for p synchronous, crash-prone, message-passing processors. The algorithm has time complexity $T = O(\log^2 p)$ and message complexity $M = O(p^{1+\varepsilon})$, for any $\varepsilon > 0$. This substantially improves the message complexity of the previous best algorithm that has $M = O(p^{1.77})$, while maintaining the same time complexity.

The strength and utility of the new result is demonstrated by constructing a deterministic algorithm for performing n tasks in this distributed setting. Previous solutions used coordinator or check-pointing approaches, immediately incurring a work penalty $\Omega(n + f \cdot p)$ for f crashes, or relied on strong communication primitives, such as reliable broadcast, or had work too close to the trivial $\Theta(p \cdot n)$

☆ This research was supported by the NSF Grants 9988304, 0121277, and 0311368. The work of the first author was performed in part at the University of Connecticut. The work of the second author was supported by the NSF-NATO Award 0209588 and by KBN grant 4T11C04425. The work of the third author was supported in part by the NSF CAREER Award 9984778. A preliminary version of this work appears as [12].

* Corresponding author.

E-mail addresses: chryssis@ucy.ac.cy (C. Georgiou), darek@mimuw.edu.pl (D.R. Kowalski), aas@cse.uconn.edu (A.A. Shvartsman).

doi:10.1016/j.tcs.2005.05.019

bound of oblivious algorithms. The new algorithm uses p crash-prone processors to perform n similar and idempotent tasks so long as one processor remains active. The work of the algorithm is $W = O(n + p \cdot \min\{f + 1, \log^3 p\})$ and its message complexity is $M = O(fp^{\varepsilon} + p \min\{f + 1, \log p\})$, for any $\varepsilon > 0$. This substantially improves the work complexity of previous solutions using simple point-to-point messaging, while "meeting or beating" the corresponding message complexity bounds.

The new algorithms use communication graphs and permutations with certain combinatorial properties that are shown to exist. The algorithms are correct for any permutations, and in particular, the same expected bounds can be achieved using random permutations.

Keywords: Distributed algorithms; Processor failures; Gossip; Performing work; Combinatorial tools

1. Introduction

The effectiveness of distributed solutions for specific problems depends on our ability to exploit parallelism in multiprocessor systems. Gathering and disseminating information in distributed settings is a key element in obtaining efficient solutions for many computation problems. The *gossip* problem is an abstraction of information propagation activity: given a set of processors where each processor initially has some piece of information, called *rumor*, the goal is to have every processor learn each rumor.

In systems of larger scale the set of processors available to a computation may dynamically change due to failures, due to processors being reassigned to other tasks, or becoming unavailable for other reasons. Thus it is necessary to design algorithms that combine efficient parallelism with the ability to tolerate perturbations in the computing medium. We consider the case where synchronous processors are subject to crashes, i.e., a processor stops and does not perform any further actions. This models both the common failure assumption and the situation where processors are reassigned to a new computation. In this setting, it may not be always possible to collect the rumor of a processor that crashes, even if some other processors learned the rumor before it crashed, since these processors may crash as well. Hence, we consider the gossip problem solved if (a) each non-faulty processor learns the rumors of all other non-faulty processors, and (b) for each crashed processor, all non-faulty processors either learn its rumor or learn that the processor crashed.

In this paper we first consider the gossip problem with p processors in the synchronous message passing system and under the adaptive adversary that can cause up to $f < p$ processor crashes. We present a new algorithm solving the gossip problem that obtains a substantially better message complexity than the previous best known solution. We demonstrate the advantage of the new algorithm by showing how to solve a standard problem of performing work in a distributed setting. Specifically, our new gossip algorithm allows us to derive a more efficient solution for the *Do-All* problem of Dwork et al. [10]: given p processors, perform n tasks in the presence of up to $f < p$ processor crashes. The *Do-All* problem is considered solved, when all tasks are performed and at least one non-faulty processor knows about this.

Background and prior results. The efficiency of algorithmic solutions to the gossip problem in synchronous message-passing models is measured in terms of time and the number of point-to-point messages. The best deterministic solution for the gossip problem under

adaptive adversaries that cause processor crashes is due to Chlebus and Kowalski [7]. Other work on the gossip problem in failure-prone settings dealt with link failures or processor failures under oblivious adversaries, or considered random failures—see the survey by Pelc [17].

A trivial solution to the gossip problem is to have every processor send its rumor to all other processors. This requires $O(1)$ time and $O(p^2)$ messages. To achieve better message complexity, Chlebus, and Kowalski [7] trade computation steps for messages. Their algorithm runs in $O(\log^2 p)$ time, sends $O(p^{1.77})$ point-to-point messages, and tolerates up to $p-1$ crashes. They also presented a lower bound for the gossip problem that states that the time has to be at least $\Omega(\log p/\log\log p)$ in order for the message complexity to be $O(p \operatorname{polylog} p)$. They also showed how to use their gossip algorithm to obtain an efficient synchronous algorithm for the *consensus* problem (processors must agree on a common value).

Algorithms for the *Do-All* problem in the message-passing models are evaluated according to the number of computation steps taken in performing the tasks (i.e., the *available processor steps* [14]), and according to their communication efficiency that accounts for all point-to-point messages. Trivial solutions to *Do-All* are obtained by having each processor obliviously perform each of the n tasks. Such solutions have work $\Theta(n \cdot p)$ and require no communication. To achieve better work efficiency we trade messages for computation steps.

Algorithms solving *Do-All* have been provided by Dwork et al. [10], by De Prisco et al. [9], and by Galil et al. [11]. (The analysis in [10] uses task-oriented work that allows processors to idle.) The algorithm by Galil et al. [11] has work $O(n+fp)$ and message complexity $O(fp^{\varepsilon}+p\ \min\{f+1, \log p\})$, where f is number of crashes. These deterministic algorithms rely on single coordinators or checkpointing. Such strategies are subject to the lower bound of $\Omega(n+(f+1)p)$ on work [9]. The algorithm of Chlebus et al. [5] beats this lower bound by using multiple coordinators. This algorithm (using the analysis in [13]) has work $O(\log f(n+p\log p/\log(p/f)))$ and message complexity $O(n+p\log p/\log(p/f)+pf)$ when $f \leqslant p/\log p$, and work $O(\log f(n+p\log p/\log\log p))$ and message complexity $O(n+p\log p/\log\log p+pf)$ when $f > p/\log p$, however it uses *reliable broadcast* (the message complexity still accounts for all point-to-point messages).

We seek solutions that obtain better work and message efficiency and that use the conventional point-to-point messaging. We see the key to such solutions in the ability to share knowledge among processors by means that are less authoritarian than the use of coordinators. Chlebus et al. [6] pursued such an approach and developed an algorithm with the combined work and message complexity of $O(n+p^{1.77})$, however, the work bound is still close to the quadratic bound obtained by oblivious algorithms.

An important aspect of *Do-All* algorithms is the sequencing of tasks done by each processor. The algorithms of Anderson and Woll [4] for the shared-memory model and of Malewicz et al. [16] for partitionable networks use approaches that provide processors with sequences of tasks based on permutations with certain combinatorial properties.

Contributions. Our objectives are to improve the efficiency of solutions for the gossip problem with p processors and to demonstrate the utility of the new solution. The first objective is achieved by providing a new solution for the gossip problem with the help of communication over expander graphs and by using permutations with specific

combinatorial properties. The second objective is met by using the gossip algorithm to solve the p-processor, n-task *Do-All* problem using an algorithmic paradigm that does not rely on coordinators, checkpointing, or reliable broadcast. Instead we use an approach where processors share information using our gossip algorithm, where the point-to-point messaging is constrained by means of a communication graph that represents a certain subset of the edges in a complete communication network. Our approach also equips processors with schedules of tasks based on permutations that we show to exist. Thus the two major contributions presented in this paper are as follows:

1. We present a new algorithm for the *gossip* problem that for p processors has time complexity $O(\log^2 p)$ and message complexity $O(p^{1+\varepsilon})$, for any $\varepsilon > 0$.
   Our algorithm substantially improves on the message complexity $M = O(p^{1.77})$ of the previously best known algorithm of Chlebus and Kowalski [7], that has the same asymptotic time complexity.
2. We demonstrate the strength and utility of our gossip algorithm by presenting a new algorithm for p processors that solves the *Do-All* problem with n tasks in the presence of any pattern of f crashes ($f < p$) with work complexity $W = O(n+p\cdot\min\{f+1, \log^3 p\})$ and message complexity $M = O(fp^{\varepsilon} + p \min\{f + 1, \log p\})$, for any $\varepsilon > 0$. The algorithm uses our new gossip algorithm as a building block to implement information sharing.
   This result improves the work complexity $W = O(n + fp)$ of the algorithm of Galil et al. [11], while obtaining the same message complexity. We also improve on the result of Chlebus et al. [6] that has $W = O(n + p^{1.77})$ and $M = O(p^{1.77})$. Unlike the algorithm of Chlebus et al. [5] that has comparable work complexity but relies on reliable broadcast, our algorithm uses simple point-to-point messaging.

The complexity analysis of our algorithms relies on permutations that we show to exist. The required permutations can be identified through exhaustive search, and it is an open problem how to construct such permutations efficiently. We show that the algorithms are correct when using arbitrary permutations, however, in that case the efficiency cannot be guaranteed. When using random permutations, then the time, work and message bounds become expected bounds. Note that when using random permutations our algorithms compare favorably to the previous randomized solutions for adaptive adversaries [7,6].

Document structure. We define the model of computation, the problems, and complexity measures in Section 2. In Section 3 we develop combinatorial tools used in the analysis of algorithms. The new gossip algorithm and its analysis is in Section 4. In Section 5 we give the new *Do-All* algorithm and its analysis. Section 6 concludes the paper with a summary of results and future work plans.

2. Models and definitions

Here we define the models, the problems we consider, and the complexity measures.

Distributed setting. We consider a system consisting of p synchronous message-passing processors; each processor has a unique identifier (pid) from the set $[p] = \{1, 2, \ldots, p\}$. We assume that p is fixed and is known to all processors. Processor activities are structured in terms of synchronous *steps*, each taking constant time.

Model of failures. We assume the *fail-stop* processor model [18], where a processor may *crash* at any moment during the computation (including within a step). Once crashed, the processor does not perform any other steps and it does not restart.

We let an omniscient *adversary* determine when to impose crashes, and we use the term *failure pattern* to denote the set of events, i.e., crashes, caused by the adversary. Following [14], we define a failure pattern F as a set of triples $\langle crash, \text{pid}, t\rangle$ 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 contains at most one triple $\langle crash, \text{pid}, t\rangle$ 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 $\text{pid} \in [p]$ *survives* step i, if for all steps $j \leqslant i$, $\langle crash, \text{pid}, j\rangle \notin F$.

For a failure pattern F, we define its *size* $|F|$ to be the number of crashes. We let f denote the maximum number of crashes that the adversary can cause. For the purpose of this paper we consider only the failure patterns with $|F| \leqslant f < p$, 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 $|F| \leqslant f < p$. The processors have no knowledge of F, $|F|$, or f (in particular, we require that algorithms must be correct for any F as long as $|F| < p$).

Communication. We assume a known upper bound on message delays. Specifically, each processor can send a message to any subset of processors in one step and the message is delivered to each (non-faulty) recipient in the next step. Messages are not corrupted and are not lost in transit. 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 do not assume reliable multicast: if a processor crashes during its multicast then an arbitrary subset of the recipients gets the message.

The Gossip *problem.* We define the *Gossip* problem as follows:

Given a set of p processors, where initially each processor has a distinct piece of information, called a rumor, *the goal is for each processor to learn all the rumors in the presence of any pattern of crashes. The following conditions must be satisfied*:

(1) *Correctness:*
   (a) *All non-faulty processors learn the rumors of all non-faulty processors*,
   (b) *For every crashed processor v, non-faulty processor w either knows that v has crashed, or w knows v's rumor*.
(2) *Termination*: *Every non-faulty processor terminates its protocol*.

We let *Gossip*(p, f) stand for the *Gossip* problem for p processors (and p rumors) and any pattern of crashes $F \in \mathcal{F}$ with $|F| \leqslant f < p$.

Tasks. We define a *task* to be a computation that can be performed by any processor in at most one time step; its execution does not 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* from the set $\mathcal{T} = [n]$. For brevity we refer to a task identifier as *tid* in the rest of the paper. We assume that n is fixed and is known to all processors.

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

Given a set $\mathcal{T}$ of n tasks, perform all tasks using p processors, in the presence of any pattern of crashes. The following conditions must be satisfied:

(1) *Correctness*: *All n tasks are completed and at least one non-faulty processor knows this*.
(2) *Termination*: *Every non-faulty processor terminates its protocol*.

We let *Do-All*(n, p, f) stand for *Do-All* for n tasks, p processors, and any pattern of crashes $F \in \mathcal{F}$ with $|F| \leqslant f < p$.

Measuring efficiency. We define the measures of efficiency used in studying the complexity of the *Gossip* and the *Do-All* problems. For the *Gossip* problem we consider *time complexity* and *message complexity*. Time complexity is measured as the number of parallel steps taken by the processors by the *termination time*, where the termination time is defined to be the first step when the correctness condition is satisfied and at least one (non-faulty) processor terminates its protocol. [1] For a given problem, if τ is the termination time of a specific execution of an algorithm, then we say that the algorithm *solves* the problem by time τ in that execution.

Definition 2.1. If a p-processor algorithm solves a problem in the presence of a failure pattern F in the failure model $\mathcal{F}$ by time $\tau(p, F)$, then its time complexity T is defined as

$$T(p, f) = \max_{F \in \mathcal{F},\ |F| \leqslant f} \{\tau(p, F)\}.$$

Message complexity is measured as the total number of point-to-point messages sent by the processors by termination time. When a processor communicates using a multicast, its cost is the resultant total number of point-to-point messages. For a p-processor computation subject to a failure pattern $F \in \mathcal{F}$, denote by $M_i(p, F)$ the number of point-to-point messages sent by the processors in step i of the computation.

Definition 2.2. If a p-processor algorithm solves a problem in the presence of a failure pattern F in the failure model $\mathcal{F}$ by time $\tau(p, F)$, then its message complexity M is defined as

$$M(p, f) = \max_{F \in \mathcal{F},\ |F| \leqslant f} \left\{ \sum_{i \leqslant \tau(p,F)} M_i(p, F) \right\}.$$

In the cases where message complexity M depends on the size of the problem n, we similarly define message complexity as $M(n, p, f)$.

In measuring work complexity, we assume that a processor performs a unit of work per unit of time. Note that the idling processors consume a unit of work per step. For a p-processor, n-task computation subject to a failure pattern $F \in \mathcal{F}$, denote by $P_i(n, p, F)$ the number of processors that survive step i of the computation.

Definition 2.3. If a p-processor algorithm solves a problem of size n in the presence of a failure pattern F in the failure model $\mathcal{F}$ by time $\tau(n, p, F)$, then its work W is

[1] The complexity results in this paper, except for the results in Section 5.4, also hold for a stronger definition of the termination time that requires that each non-faulty processor terminates its protocol.

defined as

$$W(n, p, f) = \max_{F \in \mathcal{F},\ |F| \leqslant f} \left\{ \sum_{i \leqslant \tau(n,p,F)} P_i(n, p, F) \right\}.$$

3. Combinatorial tools

We now develop the tools used in controlling the message complexity of our gossip algorithm (presented in the next section).

3.1. Communication graphs

We first describe communication graphs—conceptual data structures that constrain communication patterns in our gossip algorithm.

Informally speaking, the computation begins with a communication graph that contains all nodes, where each node represents a processor and each edge represents a communication link between processors. Each processor v can send a message to any other processor w that v considers to be non-faulty and that is a neighbor of v according to the communication graph. As processors crash, meaning that nodes are "removed" from the graph, the neighborhood of the non-faulty processors changes dynamically such that the graph induced by the remaining nodes guarantees "progress in communication": progress in communication according to a graph is achieved if there is at least one "good" connected component, which evolves suitably with time and satisfies the following properties: (i) the component contains "sufficiently many" nodes so that collectively they have learned "suitably many" rumors, (ii) it has "sufficiently small" diameter so that information can be shared among the nodes of the component without "undue delay", and (iii) the set of nodes of each successive good component is a subset of the set of nodes of the previous good component.

We use the following terminology and notation. Let $G = (V, E)$ be a (undirected) graph, with V the set of nodes (representing processors, $|V| = p$) and E the set of edges (representing communication links). We denote by G_Q the subgraph of G that it is induced by the subset Q of V. Given G_Q, we define $N_G(Q)$ to be the subset of V consisting of all the nodes in Q and their neighbors in G. The maximum node degree of graph G is denoted by Δ.

Let G_{V_i} be the subgraph of G induced by the sets V_i of nodes. Each set V_i corresponds to the set of processors that have not crashed by step i of a given computation. Hence $V_{i+1} \subseteq V_i$ (since processors do not restart). Also, each $|V_i| \geqslant p - f$, since no more than $f < p$ processors may crash in a given computation. Let G_{Q_i} denote a component of G_{V_i} where $Q_i \subseteq V_i$.

Chlebus et al. [6] formulated the notion of a "good" component (i.e., subgraph) G_{Q_i} of a subgraph G_{V_i} of graph G by setting $Q_i = P(V_i)$, where P is a witness function that is required by graph G in order to satisfy a certain property, called property $\mathcal{R}$:

Definition 3.1 (*Chlebus et al. [6]*). Graph G satisfies PROPERTY $\mathcal{R}(p, f)$ if there is a function P, which assigns subgraph $P(R) \subseteq G$ to each subgraph $R \subseteq G$ of size at least $p - f$, such that the following hold:

$\mathcal{R}.1: \ P(R) \subseteq R.$ $\mathcal{R}.3$: The diameter of $P(R)$ is at most $30 \log p + 1$.

$\mathcal{R}.2: \ |P(R)| \geqslant |R|/7.$ $\mathcal{R}.4$: If $R_1 \subseteq R_2$ then $P(R_1) \subseteq P(R_2)$.

Let $L(p, \Delta_0)$ denote the family of constructive regular graphs of p nodes and degree Δ_0, that have good expansion properties, and were introduced by Lubotzky et al. [15]. These graphs are defined and can be constructed for each number p' of the form $q(q^2 - 1)/2$, where q is a prime integer congruent to 1 modulo 4. The node degree Δ_0 can be any number such that $\Delta_0 - 1$ is a prime congruent to 1 modulo 4 and a quadratic nonresidue modulo q. It follows, from the properties of the distribution of prime numbers (see e.g. [8]), that Δ_0 can be selected to be a constant independent of p and q such that $p' = q(q^2 - 1)/2 = \Theta(p)$. Since for each p there is one such number $p' = \Theta(p)$, we let each processor simulate O(1) nodes, and we henceforth assume that p is as required so that $L(p, \Delta_0)$ can be constructed. In [6] the authors extended the result of Upfal [19], who showed that there is a function P' such that if R is a subgraph of $L(p, \Delta_0)$ of size at least $\frac{71}{72} \cdot p$ then subgraph $P'(R)$ of R has size at least $|R|/6$ and diameter at most $30 \log p$. (These constants in the case of linear-size subgraphs can be improved, see [2].) Let G^k be the kth power of graph G, that is, $G^k = (V, E')$, where the edge $(u, v) \in E'$ if and only if there is a path between u and v in G of length at most k.

In [6] the following lemma was proved.

Lemma 3.1 (*Chlebus et al. [6]*). *For every $f < p$ there exists a positive integer j such that graph $L(p, \Delta_0)^j$ has* PROPERTY $\mathcal{R}(p, f)$. *Moreover, the maximum degree Δ of graph $L(p, \Delta_0)^j$ is $O((p/p - f)^{2 \log_\rho \Delta_0})$, for some absolute*[2] *constant ρ, which for $\Delta_0 = 74$ could be taken equal to $\rho = \frac{27}{5}$.*

However, the above property is too strong for our purpose and applied to the communication analysis of our gossip algorithm does not yield the desired result. Therefore, we define a weaker property that yields the desired results with our analysis:

Definition 3.2. Graph $G = (V, E)$ has the *Compact Chain Property CCP*(p, f, ε), if:

(1) The maximum degree of G is at most $(p/p - f)^{1+\varepsilon}$,
(2) For a given sequence $V_1 \supseteq \cdots \supseteq V_k$ (where $V = V_1$), where $|V_k| \geqslant p - f$, there is a sequence $Q_1 \supseteq \cdots \supseteq Q_k$ such that for every $i = 1, \ldots, k$:
   (a) $Q_i \subseteq V_i$,
   (b) $|Q_i| \geqslant |V_i|/7$, and
   (c) the diameter of G_{Q_i} is at most $31 \log p$.

We now prove the existence of graphs satisfying the *CCP* property for some parameters.

[2] An absolute constant is a constant whose value is absolutely the same under all circumstances and for all parameters p and f.

Lemma 3.2. *For $p > 2$, every $f < p$ and constant $\varepsilon > 0$, there is a graph G of $O(p)$ nodes satisfying property $CCP(p, f, \varepsilon)$.*

Proof. Notice that for $p - f \leqslant \sqrt{p^{\varepsilon}}$, the complete graph K_p satisfies property $CCP(p, f, \varepsilon)$, for every constant $\varepsilon > 0$. The same holds if $p - f \geqslant p/4$, by applying Lemma 3.1 and setting $Q_i = P(G_i)$ (in this case Δ is constant). For the remainder of the proof we assume that $\sqrt{p^{\varepsilon}} < p - f < p/4$.

Fix f and $\varepsilon > 0$. Our candidate for graph G is a graph $L(p, \Delta)$, where we take the smallest possible $\Delta \geqslant 9 + (p/(p-f))^{1+\varepsilon}$. (By properties of graphs L, we can find $\Delta = O(1 + (p/(p-f))^{1+\varepsilon})$.) Let $\lambda = 2\sqrt{\Delta - 1}$ be the bound for the absolute value of the second eigenvalue of graph $L(p, \Delta)$ (see [15]). Alon and Chung [1] showed that for every set $R \subseteq V$, the number of edges in the subgraph induced by R (denoted by $e(R)$) can be bounded as follows:

$$\left| e(R) - \frac{\Delta \cdot |R|^2}{2p} \right| \leqslant \frac{\lambda}{2} \left(1 - \frac{|R|}{p} \right) |R|. \tag{1}$$

For a given graph induced by R such that $\sqrt{p^{\varepsilon}} < |R| < p/4$ and a subset $Q \subseteq R$, we denote by $\mathcal{S}_{Q,R}$ the family of sets $S \supseteq Q$ such that S is a maximal (in the sense of inclusion) subset of R such that no node in S has more than $\Delta(|R|/2p)$ neighbors outside of S in graph G. We call a subgraph induced by S a *simple expander*, if for every $S' \subseteq S$ of size at most $|S|/2$, $|N_S(S')| \geqslant 4|S'|/3$. We assume that Q is a simple expander that has size less than $|R|/7$.

Claim. *For $p > 2$, if $\sqrt{p^{\varepsilon}} < |R| < p/4$ then for every subset $S \in \mathcal{S}_{Q,R}$, S is of size $|R|/7$ and a subgraph induced by S is a simple expander. Hence a diameter of the subgraph induced by S is at most $4 \log p$.*

We prove the claim. Consider any $S \in \mathcal{S}_{Q,R}$. First we show that $|S| \geqslant |R|/7$. Suppose to the contrary, that $|S| < |R|/7$. By applying inequality (1) and setting $\Delta > 9$ and $\lambda = 2\sqrt{\Delta - 1}$, we obtain that

$$\begin{aligned} e(V \setminus S) &\leqslant \frac{\Delta\,(p - |S|)^2}{2p} + \frac{\lambda |S|}{2p}\,(p - |S|) \\ &\leqslant \frac{\Delta\,(p - |S|)}{2} - \frac{\Delta\,(p - |S|)}{2} \frac{|S|}{p} + \frac{\Delta\,(p - |S|)}{3} \frac{|S|}{p} \\ &= \frac{\Delta\,(p - |S|)}{2} - \frac{\Delta\,(p - |S|)}{6} \frac{|S|}{p} < \frac{\Delta\,(p - |S|)}{2} - \frac{\Delta\,(p - (|R|/7))}{6} \frac{|S|}{p}. \end{aligned}$$

This contradicts the definition of S, since from the definition of S it follows that the number of edges having one end in S and the other end outside of S, is at most $\Delta |R||S|/2p$, and consequently

$$e(V \setminus S) \geqslant \frac{\Delta\,(p - |S|)}{2} - \frac{\Delta |R||S|}{4p} > \frac{\Delta\,(p - |S|)}{2} - \frac{\Delta\,(p - (|R|/7))}{6} \frac{|S|}{p}.$$

Next we show that for every $S' \subseteq S$ of size at most $|S|/2$, we have $|N_S(S')| \geqslant 4|S'|/3$. By definition of S, the total number of edges incident to nodes in S' is at least

$|S'|\Delta(1-(|R|/2p))$. On the other hand, using inequality (1) we obtain

$$e(S') \leqslant \frac{\Delta \cdot |S'|^2}{2p} + \frac{\lambda}{2}\left(1 - \frac{|S'|}{p}\right)|S'|.$$

Thus the number of edges having one end in S' and the other end outside of S' is at least

$$\begin{aligned} |S'|\Delta\left(1-\frac{|R|}{2p}\right) - e(S') &\geqslant |S'|\Delta\left(1-\frac{|R|}{2p}\right) - \frac{\Delta\cdot|S'|^2}{2p} - \frac{\lambda}{2}\left(1-\frac{|S'|}{p}\right)|S'| \\ &\geqslant |S'|\Delta\cdot\left(1-\frac{|R|+|S'|}{2p} - \frac{1}{\sqrt{\Delta+1}}\right) \\ &\geqslant |S'|\Delta/3. \end{aligned}$$

Since every node in $N_S(S')\backslash S'$ has at most Δ neighbors in S', it follows that $|N_S(S')\backslash S'| \geqslant (|S'|\Delta/3)/\Delta = |S'|/3$. Consequently S is a simple expander. We show that the diameter of S is at most $2\log_{3/2} p < 4\log p$. Consider two nodes $v, w \in S$. By the simple-expansion property, the number $N_{S^{\log_{3/2} p}}(v)$ (and also $N_{S^{\log_{3/2} p}}(w)$) of nodes of distance $\log_{3/2} p$ from v (and also from w) in the graph induced by S is greater than $p/2$. Consequently $N_{S^{\log_{3/2} p}}(v) \cap N_{S^{\log_{3/2} p}}(w) \neq \emptyset$, and then the shortest path between v and w is of length at most $2\log_{3/2} p < 4\log p$. This completes the proof of the claim.

We now show how to construct a sequence $Q_1 \supseteq \cdots \supseteq Q_k$ for a sequence $V_1 \supseteq \cdots \supseteq V_k$, so that property $CCP(p, f, \varepsilon)$ is satisfied. We proceed inductively: we apply the claim to the set $R = V_k$ and define Q_k to be a set from $\mathcal{S}_{V_k,\emptyset}$. If we have defined set Q_i, for $1 < i \leqslant k$, we apply the claim to the set $R = V_{i-1}$ and define Q_{i-1} to be a set in $\mathcal{S}_{V_{i-1},Q_i}$ including set Q_i. The inductive proof shows that the Q_is are well defined and that graph G satisfies property $CCP(p, f, \varepsilon)$. More precisely, the following invariant holds after construction of set Q_i:

(a) $Q_i \subseteq V_i$ and $Q_i \supseteq \cdots \supseteq Q_k$,
(b) $|Q_i| \geqslant |V_i|/7$,
(c) the diameter of G_{Q_i} is at most $31\log p$,
(d) every node in Q_i has at most $\Delta(|R|/2p)$ neighbors outside of Q_i in graph G.

We show that for $i > 1$ the set Q_{i-1} is well defined and satisfies the invariant. For $i = k$ it follows directly from the claim. Consider $1 < i < k$. From property (d) in the invariant after step i it follows that if we apply the claim to the set $R = V_{i-1}$ then Q_i is included in some $S \in \mathcal{S}_{V_{i-1},Q_i}$. Consequently the definition of Q_{i-1} is correct. By the thesis of the claim applied to such R and S we obtain properties (b) and (c) of invariant after step $i-1$. Properties (a) and (d) follow directly from the definition of Q_{i-1}. □

Remark 3.1. In the presentation up to this point we included among the properties of interest the property that $Q_1 \supseteq \cdots \supseteq Q_k$. We believe that it is a potentially valuable property of independent interest that can be used in the analysis of fault-tolerant algorithms. As it happens with our algorithms in this paper, we can obtained the needed results without resorting to using this chain property.

3.2. Sets of permutations and their properties

We now deal with *sets of permutations* that satisfy *certain properties*. These permutations are used by the processors in the gossip algorithm to decide to what subset of processors they send their rumor in each step of a given execution. Consider the group S_t of all permutations on set $\{1, \ldots, t\}$, with the composition operation $\circ$, and identity $\mathbf{e}_t$ (t is a positive integer). For permutation $\pi = \langle \pi(1), \ldots, \pi(t) \rangle$ in S_t, we say that $\pi(i)$ is a d-left-to-right maximum (d-lrm in short), if there are less than d previous elements in π of value greater than $\pi(i)$, i.e., $|\{\pi(j) : \pi(j) > \pi(i) \wedge j < i\}| < d$.

Let Υ and Ψ, $\Upsilon \subseteq \Psi$, be two sets containing permutations from S_t. For every σ in S_t, let $\sigma \circ \Upsilon$ denote the set of permutation $\{\sigma \circ \pi : \pi \in \Upsilon\}$. For a given permutation π, let (d)-$LRM(\pi)$ denote the number of d-left-to-right maxima in π. Now we define the notion of *surfeit*.[3] For a given Υ and permutation $\sigma \in S_t$, let $(d, |\Upsilon|)$-Surf(Υ, σ) be equal to $\sum_{\pi \in \Upsilon} (d)$-$LRM(\sigma^{-1} \circ \pi)$. We then define the (d, q)-surfeit of set Ψ as (d, q)-Surf$(\Psi) =$ $\max\{(d, q)\text{-Surf}(\Upsilon, \sigma) : \Upsilon \subseteq \Psi \wedge |\Upsilon| = q \wedge \sigma \in S_t\}$. Finally, we let H_x denote the Harmonic function of x, where x is a positive natural number.

We obtain the following results for (d, q)-surfeit.

Lemma 3.3. *Let Υ be a set of q random permutations from S_t. For every fixed positive integer d, the probability of event (d, q)-Surf$(\Upsilon, \mathbf{e}_t) > t \ln t + 10qd \ln(t + p)$ is at most* $\mathrm{e}^{-[t \ln t + 9qdH_{t+p}] \ln(9/e)}$.

Proof. First observe, that for $d \geqslant t/e$ the thesis is obvious. In the rest of the proof we assume $d < t/e$.

First we describe the way of generating random permutations. This is done by induction on the number of elements $i \leqslant t$ that are permuted. When $i = 1$, there is only one permutation and this permutation is random. Suppose we can generate random permutation of $i - 1$ different elements, we show how to permute i elements. First we choose randomly one element from the i elements and put it as the last element in the permutation. By induction we generate a random permutation from the remaining $i - 1$ elements and we put these elements as the first $i - 1$ elements in the permutation. Simple induction proof shows that every permutation of i elements has equal probability, since it is a concatenation of two independent and random events.

It follows that the random set of permutation Υ can be selected by applying the above rule q times, independently. Let $X(\pi, i)$, for $i = 1, \ldots, t$, be the random value such that $X(\pi, i) = 1$ if $\pi(i)$ is a d-lrm in π, and $X(\pi, i) = 0$ otherwise.

Claim. *Using the above method of generating random permutation we can show that if π is a random permutation, then $X(\pi, i) = 1$ with probability* $\min\{d/i, 1\}$, *independently of other values $X(\pi, j)$, for $j > i$. More precisely,* $\Pr[X(\pi, i) = 1 | \bigwedge_{j>i} X(\pi, j) = a_j] = \min\{d/i, 1\}$, *for any* 0–1 *sequence $a_{i+1}, \ldots, a_t$.*

[3] We will show that *surfeit* relates to the redundant activity in our algorithms, i.e., "overdone" activity, or literally "surfeit".

This is because $\pi(i)$ might be a d-lrm if during the $(t-i-1)$th step of the generation of π we selected randomly one of the d greatest remaining elements (there are $i \geqslant d$ remaining elements in this step of generation; if $i = d$, then by definition $\pi(i)$ is a d-lrm with probability one). Hence the claim is proved.

First notice that for every $\pi \in \Upsilon$ and every $i = 1, \ldots, d$, $\pi(i)$ is d-lrm. Second, observe that $\mathbb{E}\left[\sum_{\pi\in\Upsilon}\sum_{i=d+1}^{t} X(\pi, i)\right] = qd \cdot \sum_{i=d+1}^{t} \frac{1}{i} = qd(H_t - H_d)$. We use Chernoff bound (see [3])

$$\Pr\left[\sum_j Y_j > \mathbb{E}\left[\sum_j Y_j\right](1+b)\right] < \left(\frac{e^b}{(1+b)^{1+b}}\right)^{\mathbb{E}[\sum_j Y_j]} < e^{-\mathbb{E}[\sum_j Y_j](1+b)\ln(1+b)/e},$$

where Y_j are independent random 0–1 variables and $b > 0$ is any constant, to prove the lemma.

We use Chernoff bound and derive the following (for some $p < t$):

$$\begin{aligned}
&\Pr\left[\sum_{\pi\in\Upsilon}\sum_{i=d+1}^{t} X(\pi, i) > t\ln t + 9qdH_{t+p}\right] \\
&= \Pr\left[\sum_{\pi\in\Upsilon}\sum_{i=d+1}^{t} X(\pi, i) > qd(H_t - H_d) \cdot \frac{t\ln t + 9qdH_{t+p}}{qd(H_t - H_d)}\right] \\
&\leqslant e^{-qd(H_t-H_d)\frac{t\ln t+9qdH_{t+p}}{qd(H_t-H_d)}\ln\frac{t\ln t+9qdH_{t+p}}{e\cdot qd(H_t-H_d)}} \\
&\leqslant e^{-[t\ln t+9qdH_{t+p}]\ln(9/e)}
\end{aligned}$$

since $\frac{t\ln t+9qdH_{t+p}}{qd(H_t-H_d)} > 1$ (the condition for using Chernoff bound of this type).

From the above and the fact that $\ln i \leqslant H_i \leqslant \ln i + 1$, we obtain that

$$\begin{aligned}
&\Pr\left[\sum_{\pi\in\Upsilon}\sum_{i=1}^{t} X(\pi, i) > t\ln t + 10qd\ln(t+p)\right] \\
&\leqslant \Pr\left[\sum_{\pi\in\Upsilon}\sum_{i=d+1}^{t} X(\pi, i) > t\ln t + 9qdH_{t+p}\right] \\
&\leqslant e^{-[t\ln t+9qdH_{t+p}]\ln(9/e)}.
\end{aligned}$$

This completes the proof of the lemma. □

Theorem 3.4. *Let Ψ be a set of p random permutations from S_t. The probability of event "there are positive integers d and $q \leqslant p$, (d, q)-Surf$(\Psi) > t\ln t + 10qd\ln(t+p)$" is at most* $e^{-t\ln t\cdot\ln(9/e^2)}$.

Proof. Observe that for $d \geqslant t/e$ the result is straightforward. In the rest of the proof we assume that $d < t/e$.

First notice, that if Υ is a random set of permutations, then for an arbitrary permutation σ on the set $\{1, \ldots, t\}$, the set $\sigma^{-1} \circ \Upsilon$ is also a random set of permutations, since contraction with permutation is a bijective operation on sets of q permutations. Consequently,

by Lemma 3.3, (d,q)-Surf$(\Upsilon,\sigma) > t\ \ln t + 10qd\ln(t+p)$ holds with probability at most $\mathrm{e}^{-[t\ \ln t+9qdH_{t+p}]\ \ln(9/e)}$.

Hence the probability that a random set Ψ of p permutations satisfies (d,q)-Surf$(\Psi) > t\ \ln t + 10qd\ln(t+p)$ is at most

$$t!\cdot\binom{p}{q}\cdot \mathrm{e}^{-[t\ \ln t+9qdH_{t+p}]\ \ln(9/e)} \leqslant \mathrm{e}^{t\ \ln t+q\ \ln(ep/q)-[t\ \ln t+9qdH_{t+p}]\ \ln(9/e)} \leqslant \mathrm{e}^{-[t\ \ln t+8qdH_{t+p}]\ \ln(9/\mathrm{e}^2)}.$$

It follows, that the probability of event:

"there are positive integers d and q such that (d,q)-Surf$(\Psi) > t\ \ln t + 10qd\ln(t+p)$", is at most

$$\sum_{d=1}^{\lceil t/e\rceil-1}\sum_{q=1}^{p}\mathrm{e}^{-[t\ \ln t+8qdH_{t+p}]\ln(9/\mathrm{e}^2)}\sum_{d=\lceil t/e\rceil}^{\infty}\sum_{q=1}^{p}0\leqslant\mathrm{e}^{-t\ \ln t\cdot\ln(9/\mathrm{e}^2)},$$

for $p\geqslant 1$ and $t\geqslant 3$. □

Using the probabilistic method [3] we obtain the following result.

Corollary 3.5. *There is a set of p permutations Ψ from S_t such that, for every positive integers d and $q\leqslant p$, (d,q)-Surf$(\Psi)\leqslant t\ \ln t + 10qd\ln(t+p)$.*

The efficiency of our gossip algorithm relies on the existence of the permutations in the thesis of the corollary (however the algorithm is correct for any permutations).

4. The gossip algorithm

Our new gossiping algorithm, called Gossip$_\varepsilon$, improves on the algorithm in [7]. The improvement is obtained by using the better properties of communication graphs described in Lemma 3.2, the set of permutations with certain properties stated in Corollary 3.5, and by using many epochs instead of the two epochs in [7] (epochs are referred to as phases in [7]). Moreover, the communication graphs we consider have dynamically changing degree, as opposed to [7] where the authors consider graphs with fixed degree. The challenges motivating our techniques are: (i) how to assure low communication during every epoch, and (ii) how to switch between epochs without a "huge complexity hit".

4.1. Description of algorithm Gossip$_\varepsilon$

Suppose a constant ε is given such that $0 < \varepsilon < 1/3$. The algorithm proceeds in a loop that is repeated until each non-faulty processor v learns either the rumor of every processor w or that w has crashed. A single iteration of the loop is called an *epoch*. The algorithm terminates after $\lceil 1/\varepsilon\rceil - 1$ epochs. Each of the first $\lceil 1/\varepsilon\rceil - 2$ epochs consists of $\alpha\log^2 p$ *phases*, where α is such that $\alpha\log^2 p$ is the smallest integer that is larger than $341\log^2 p$. Each phase is divided into two *stages*, the *update* stage, and the *communication* stage.

Initialization

$status_v = \mathsf{collector}$;
$\text{ACTIVE}_v = \langle 1, 2, \ldots, p\rangle$;
$\text{BUSY}_v = \langle \pi_v(1), \pi_v(2), \ldots, \pi_v(p)\rangle$;
$\text{WAITING}_v = \langle \pi_v(1), \pi_v(2), \ldots, \pi_v(p)\rangle \backslash \langle v\rangle$;
$\text{RUMORS}_v = \langle (v, rumor_v)\rangle$;
$\text{NEIGHB}_v = N_{G_1}(v)\backslash\{v\}$;
$\text{CALLING}_v = \{\}$;
$\text{ANSWER}_v = \{\}$;

Gossip Algorithm

for $\ell = 1$ **to** $\lceil 1/\varepsilon\rceil - 2$ **do** *% iterating for each epoch ℓ*

  if BUSY_v is empty **then** set $status_v$ to idle;
  $\text{NEIGHB}_v = \{w : w \in \text{ACTIVE}_v \wedge w \in N_{G_\ell}(v)\backslash\{v\}\}$;
  repeat $\alpha \log^2 p$ times *% iterating phases*
    update stage; *% see description in Section 4.1.3*
    communication stage; *% see description in Section 4.1.2*

update stage; *% Terminating epoch begins*
if $status_v = \mathsf{collector}$ **then**
  send $\langle \text{ACTIVE}_v, \text{BUSY}_v, \text{RUMORS}_v, \mathsf{call}\rangle$ to each processor in WAITING_v;
receive messages;
send $\langle \text{ACTIVE}_v, \text{BUSY}_v, \text{RUMORS}_v, \mathsf{reply}\rangle$ to each processor in ANSWER_v;
receive messages;
update RUMORS_v; *% see description in Section 4.1.4*

Fig. 1. Algorithm $\text{GOSSIP}_\varepsilon$, stated for processor v; $\pi_v(i)$ denotes the ith element of permutation π_v.

In the update stage processors update their local knowledge regarding other processors' rumor (known/unknown) and condition (crashed/operational) and in the communication stage processors exchange their local knowledge (more momentarily). We say that processor v *heard about processor* w if either v knows the rumor of w or it knows that w has crashed. Epoch $\lceil 1/\varepsilon\rceil - 1$ is the terminating epoch where each processor sends a message to all the processors that it has not been heard about, requesting their rumor.

The pseudocode for the algorithm is given in Fig. 1. The details of the algorithm are explained in the rest of this section. (In the code we assume, where needed, that every *if–then* has an implicit *else* clause containing the necessary number of no-ops to match the length of the code in the *then* clause; this is used to ensure the synchrony of the system.)

4.1.1. Local knowledge and messages

Initially each processor v has its $rumor_v$ and permutation π_v from a set Ψ of permutations on $[p]$, such that Ψ satisfies the thesis of Corollary 3.5. Moreover, each processor v is associated with the variable $status_v$. Initially $status_v = \mathsf{collector}$ (and we say that v is a collector), meaning that v has not heard from all processors yet. Once v hears from all other processors, then $status_v$ is set to informer (and we say that v is an informer), meaning

that now v will inform the other processors of its status and knowledge. When processor v learns that all non-faulty processors w also have $status_w = \mathsf{informer}$ then at the beginning of the next epoch, $status_v$ becomes idle (and we say that v idles), meaning that v idles until termination, but it might send responses to messages (see call-messages below).

Each processor maintains several lists and sets. We now describe the lists maintained by processor v:

- List ACTIVE_v: it contains the pids of the processors that v considers to be non-faulty. Initially, list ACTIVE_v contains all p pids.
- List BUSY_v: it contains the pids of the processors that v consider as collectors. Initially list BUSY_v contains all pids, *permuted according to* π_v.
- List WAITING_v: it contains the pids of the processors that v did not hear from. Initially list WAITING_v contains all pids except from v, *permuted according to* π_v.
- List RUMORS_v: it contains pairs of the form $(w, rumor_w)$ or $(w, \perp)$. The pair $(w, rumor_w)$ denotes the fact that processor v knows processor w's rumor and the pair $(w, \perp)$ means that v does not know w's rumor, but it knows that w has crashed. Initially list RUMORS_v contains the pair $(v, rumor_v)$.

(Note that the lists ACTIVE and RUMORS are used as sets. Since the other structures need to be lists, for presentation purposes we adopt the convention that list structures are used for collecting information, while set structures are used for sending information.)

A processor can send a message to any other processor, but to lower the message complexity, in some cases (see communication stage) we require processors to communicate according to a conceptual communication graph G_ℓ, $\ell \leqslant \lceil 1/\varepsilon \rceil - 2$, that satisfies property $CCP(p, p - p^{1-\ell\varepsilon}, \varepsilon)$ (see Definition 3.2 and Lemma 3.2). When processor v sends a message m to another processor w, m contains lists ACTIVE_v, BUSY_v, RUMORS_v, and the variable *type*. When $type = \mathsf{call}$, processor v requires an answer from processor w and we refer to such message as a *call-message*. When $type = \mathsf{reply}$, no answer is required—this message is sent as a response to a call-message.

We now present the sets maintained by processor v:

- Set NEIGHB_v: it contains the pids of the processors that are in ACTIVE_v and that according to the communication graph G_ℓ, for a given epoch ℓ, are neighbors of v ($\text{NEIGHB}_v = \{w : w \in \text{ACTIVE}_v \wedge w \in N_{G_\ell}(v) \backslash \{v\}\}$). Initially, NEIGHB_v contains all neighbors of v (all nodes in $N_{G_1}(v) \backslash \{v\}$).
- Set CALLING_v: it contains the pids of the processors that v will send a call-message. Initially CALLING_v is empty.
- Set ANSWER_v: it contains the pids of the processors that v received a call-message. Initially set ANSWER_v is empty.

4.1.2. Communication stage

In this stage the processors communicate in an attempt to obtain information from other processors. This stage contains *four sub-stages*:

- First sub-stage: every processor v that is either a collector or an informer (i.e., $status_v \neq \mathsf{idle}$) sends message $\langle \text{ACTIVE}_v, \text{BUSY}_v, \text{RUMORS}_v, \mathsf{call} \rangle$ to every processor in CALLING_v. The idle processors do not send any messages in this sub-stage.
- Second sub-stage: all processors (collectors, informers and idling) collect the information sent to by the other processors in the previous sub-stage. Specifically, processor v collects

lists ACTIVE$_w$, BUSY$_w$ and RUMORS$_w$ of every processor w that received a call-message from and v inserts w in set ANSWER$_v$.

- Third sub-stage: every processor (regardless of its status) responds to each processor that received a call-message from. Specifically, processor v sends message $\langle$ACTIVE$_v$, BUSY$_v$, RUMORS$_v$, reply$\rangle$ to the processors in ANSWER$_v$ and empties ANSWER$_v$.
- Fourth sub-stage: the processors receive the responses to their call-messages.

4.1.3. Update stage

In this stage each processor v updates its local knowledge based on the messages it received in the *last communication stage*.[4] If $status_v = \mathsf{idle}$, then v idles. We now present the six *update rules* and their processing. Note that the rules are not disjoint, and we apply them in the order from (r1) to (r6):

(r1) Updating BUSY$_v$ or RUMORS$_v$: For every processor w in CALLING$_v$ (i) if v is an informer, it removes w from BUSY$_v$, (ii) if v is a collector and RUMORS$_w$ was included in one of the messages that v received, then v adds the pair $(w, rumor_w)$ in RUMORS$_v$ and, (iii) if v is a collector but RUMORS$_w$ was not included in one of the messages that v received, then v adds the pair $(w, \perp)$ in RUMORS$_v$.

(r2) Updating RUMORS$_v$ and WAITING$_v$: For every processor w in $[p]$, (i) if $(w, rumor_w)$ is not in RUMORS$_v$ and v learns the rumor of w from some other processor that received a message from, then v adds $(w, rumor_w)$ in RUMORS$_v$ (updating RUMORS$_v$), (ii) if both $(w, rumor_w)$ and $(w, \perp)$ are in RUMORS$_v$, then v removes $(w, \perp)$ from RUMORS$_v$ (updating RUMORS$_v$), (iii) if either of $(w, rumor_w)$ or $(w, \perp)$ is in RUMORS$_v$ and w is in WAITING$_v$, then v removes w from WAITING$_v$ (updating WAITING$_v$).

(r3) Updating BUSY$_v$: For every processor w in BUSY$_v$, if v receives a message from processor v' so that w is not in BUSY$_{v'}$, then v removes w from BUSY$_v$.

(r4) Updating ACTIVE$_v$ and NEIGHB$_v$: For every processor w in ACTIVE$_v$ (i) if w is not in NEIGHB$_v$ and v received a message from processor v' so that w is not in ACTIVE$_{v'}$, then v removes w from ACTIVE$_v$, (ii) if w is in NEIGHB$_v$ and v did not receive a message from w, then v removes w from ACTIVE$_v$ and NEIGHB$_v$, and (iii) if w is in CALLING$_v$ and v did not receive a message from w, then v removes w from ACTIVE$_v$.

(r5) Changing status: If the size of RUMORS$_v$ is equal to p and v is a collector, then v becomes an informer.

(r6) Updating CALLING$_v$: Processor v empties CALLING$_v$ and (i) if v is a collector then it updates set CALLING$_v$ to contain the first $p^{(\ell+1)\varepsilon}$ pids of list WAITING$_v$ (or all pids of WAITING$_v$ if *sizeof*(WAITING$_v$) $< p^{(\ell+1)\varepsilon}$) and all pids of set NEIGHB$_v$, and (ii) if v is an informer then it updates set CALLING$_v$ to contain the first $p^{(\ell+1)\varepsilon}$ pids of list BUSY$_v$ (or all pids of BUSY$_v$ if *sizeof*(BUSY$_v$) $< p^{(\ell+1)\varepsilon}$) and all pids of set NEIGHB$_v$.

4.1.4. Terminating epoch

Epoch $\lceil 1/\varepsilon \rceil - 1$ is the last epoch of the algorithm. In this epoch, each processor v updates its local information based on the messages it received in the last communication stage of epoch $\lceil 1/\varepsilon \rceil - 2$. If after this update processor v is still a collector, then it sends a call-message

[4] In the first update stage of the first phase of epoch 1, where no communication has yet to occur, no update of the lists or sets takes place.

to every processor that is in WAITING_v (list WAITING_v contains the pids of the processors that v does not know their rumor, or does not know whether they have crashed). Then every processor v receives the call-messages sent by the other processors (set ANSWER_v is updated to include the senders). Next, every processor v that received a call-message sends its local knowledge to the sender (i.e. to the members of set ANSWER_v). Finally each processor v updates RUMORS_v based on any received information. More specifically, if a processor w responded to v's call-message (meaning that v now learns the rumor of w), then v adds (w, rumor_w) in RUMORS_v. If w did not respond to v's call-message, and (w, rumor_w) is not in RUMORS_v (it is possible for processor v to learn the rumor of w from some other processor v' that learned the rumor of w before processor w crashed), then v knows that w has crashed and adds $(w, \perp)$ in RUMORS_v.

4.2. *Correctness of algorithm* $\text{GOSSIP}_\varepsilon$

We show that algorithm $\text{GOSSIP}_\varepsilon$ solves the $\mathit{Gossip}(p, f)$ problem correctly, meaning that by the end of epoch $\lceil 1/\varepsilon \rceil - 1$ each non-faulty processor has heard about all other $p - 1$ processors. First we show that no non-faulty processor is removed from a processor's list of active processors.

Lemma 4.1. *In any execution of algorithm* $\text{GOSSIP}_\varepsilon$, *if processors* v *and* w *are non-faulty by the end of any epoch* $\ell < \lceil 1/\varepsilon \rceil - 1$, *then* w *is in* ACTIVE_v.

Proof. Consider processors v and w that are non-faulty by the end of epoch $\ell < \lceil 1/\varepsilon \rceil - 1$. We show that w is in ACTIVE_v. The proof of the inverse is done similarly. The proof proceeds by induction on the number of epochs.

Initially all processors (including w) are in ACTIVE_v. Consider phase s of epoch 1 (for simplicity assume that s is not the last phase of epoch 1). By the update rule, a processor w is removed from ACTIVE_v if v is not idle and

(a) during the communication stage of phase $(s - 1)$, w is not in NEIGHB_v and v received a message from a processor v' so that w is not in $\text{ACTIVE}_{v'}$,
(b) during the communication stage of phase $(s - 1)$, w is in NEIGHB_v and v did not receive a message from w, or
(c) v sent a call-message to w in the communication stage of phase $(s - 1)$ of epoch 1 and v did not receive a response from w in the same stage.

Case (c) is not possible: Since w is non-faulty in all phases s' of epoch 1, w receives the call-message from v in the communication stage of phase $(s - 1)$ and adds v in ANSWER_w. Then, processor w sends a response to v in the same stage. Hence v does not remove w from ACTIVE_v. Case (b) is also not possible: Since w is non-faulty and w is in NEIGHB_v, by the properties of the communication graph G_1, v is in NEIGHB_w as well (and since v is non-faulty). From the description of the first sub-stage of the communication stage, if $\mathit{status}_w \neq \mathsf{idle}$, w sends a message to its neighbors, including v. If $\mathit{status}_w = \mathsf{idle}$, then w will not send a message to v in the first sub-stage, but it will send a reply to v' call-message in the third sub-stage. Therefore, by the end of the communication stage, v has received a message from w and hence it does not remove w from ACTIVE_v. Case (a) is also not possible: This follows inductively, using points (b) and (c): no processor will remove w from its set

of active processors in a phase prior to s and hence v does not receive a message from any processor v' so that w is not in $\text{ACTIVE}_{v'}$.

Now, assuming that w is in ACTIVE_v by the end of epoch $\ell - 1$, we show that w is still in ACTIVE_v by the end of epoch ℓ. Since w is in ACTIVE_v by the end of epoch $\ell - 1$, w is in ACTIVE_v at the beginning of the first phase of epoch ℓ. Using similar arguments as in the base case of the induction and from the inductive hypothesis, it follows that w is in ACTIVE_v by the end of the first phase of epoch ℓ. Inductively it follows that w is in ACTIVE_v by the end of the last phase of epoch ℓ, as desired. □

Next we show that if a non-faulty processor w has not heard from all processors yet then no non-faulty processor v removes w from its list of busy processors.

Lemma 4.2. *In any execution of algorithm* $\text{GOSSIP}_\varepsilon$ *and any epoch* $\ell < \lceil 1/\varepsilon \rceil - 1$, *if processors* v *and* w *are non-faulty by the end of epoch* ℓ *and* $status_w = \mathsf{collector}$, *then* w *is in* BUSY_v.

Proof. Consider processors v and w that are non-faulty by the end of epoch $\ell < \lceil 1/\varepsilon \rceil - 1$ and $status_w = \mathsf{collector}$. The proof proceeds by induction on the number of epochs.

Initially all processors w have status collector and w is in BUSY_v ($\text{CALLING}_v \setminus \text{NEIGHB}_v$ is empty). Consider phase s of epoch 1. By the update rule, a processor w is removed from BUSY_v if

(a) at the beginning of the update stage of phase s, v is an informer and w is in CALLING_v, or
(b) during the communication stage of phase s, v receives a message from a processor v' where w is not in $\text{BUSY}_{v'}$.

Case (a) is not possible: Since v is an informer and w is in CALLING_v at the beginning of the update stage of phase s, this means that in the communication stage of phase $s - 1$, processor v was already an informer and it sent a call-message to w. In this case, w would receive this message and it would become an informer during the update stage of phase s. This violates the assumption of the lemma.

Case (b) is also not possible: For w not being in $\text{BUSY}_{v'}$ it means that either

(i) in some phase $s' < s$, processor v' became an informer and sent a call-message to w, or
(ii) during the communication stage of a phase $s'' < s$, v' received a message from a processor v'' so that w was not in $\text{BUSY}_{v''}$.

Case (i) implies that in phase $s' + 1$, processor w becomes an informer which violates the assumption of the lemma. Using inductively case (i) it follows that case (ii) is not possible either.

Now, assuming that by the end of epoch $\ell - 1$, w is in BUSY_v we would like to show that by the end of epoch ℓ, w is still in BUSY_v. Since w is in BUSY_v by the end of epoch $\ell - 1$, w is in BUSY_v at the beginning of the first phase of epoch ℓ. Using similar arguments as in the base case of the induction and from the inductive hypothesis, it follows that w is in BUSY_v by the end of the first phase of epoch ℓ. Inductively it follows that w is in BUSY_v by the end of the last phase of epoch ℓ, as desired. □

We now show that each processor's list of rumors is updated correctly.

Lemma 4.3. *In any execution of algorithm* GOSSIP$_\varepsilon$ *and any epoch* $\ell < \lceil 1/\varepsilon \rceil - 1$,
(i) *if processors* v *and* w *are non-faulty by the end of epoch* ℓ *and* w *is not in* WAITING$_v$, *then* $(w, rumor_w)$ *is in* RUMORS$_v$, *and*
(ii) *if processor* v *is non-faulty by the end of epoch* ℓ *and* $(w, \perp)$ *is in* RUMORS$_v$, *then* w *is not in* ACTIVE$_v$.

Proof. We first prove part (i) of the lemma. Consider processors v and w that are non-faulty by the end of epoch ℓ and that w is not in WAITING$_v$. The proof proceeds by induction on the number of epochs. The proof for the first epoch is done similarly as the proof of the inductive step (that follows), since at the beginning of the computation each $w \neq v$ is in WAITING$_v$ and RUMORS$_v$ contains only the pair $(v, rumor_v)$, for every processor v.

Assume that part (i) of the lemma holds by the end of epoch $\ell - 1$, we would like to show that it also holds by the end of epoch ℓ. First note the following facts: no pair of the form $(w, rumor_w)$ is ever removed from RUMORS$_v$ and no processor identifier is ever added to WAITING$_v$. We use these facts implicitly in the remainder of the proof (cases (a) and (b)). Suppose, to the contrary, that at the end of epoch ℓ there are processors v, w which are non-faulty by the end of epoch ℓ and w is not in WAITING$_v$ and $(w, \perp)$ is in RUMORS$_v$. Take v such that v put the pair $(w, \perp)$ to its RUMORS$_v$ as the earliest node during epoch ℓ and this pair has remained in RUMORS$_v$ by the end of epoch ℓ. It follows that during epoch ℓ at least one of the following cases must have happened:
(a) Processor v sent a call-message to processor w in the communication stage of some phase and v did not receive a response from w (see update rule (r1)). But since w is not-faulty by the end of epoch ℓ it replied to v according to the third sub-stage of communication stage. This is a contradiction.
(b) During the communication stage of some phase processor v received a message from processor v' so that $(w, \perp)$ is in RUMORS$_{v'}$ (see update rule (r2)). But this contradicts the choice of v.
Hence part (i) is proved.

The proof of part (ii) of the lemma is analogous to the proof of part (i). The key argument is that the pair $(w, \perp)$ is added in RUMORS$_v$ if w does not respond to a call-message sent by v which in this case w is removed from ACTIVE$_v$ (if w was not removed from ACTIVE$_v$ earlier). □

Finally we show the correctness of algorithm GOSSIP$_\varepsilon$.

Theorem 4.4. *By the end of epoch* $\lceil 1/\varepsilon \rceil - 1$ *of any execution of algorithm* GOSSIP$_\varepsilon$, *every non-faulty processor* v *either knows the rumor of processor* w *or it knows that* w *has crashed.*

Proof. Consider a processor v that is non-faulty by the end of epoch $\lceil 1/\varepsilon \rceil - 1$. Note that the claims of Lemmas 4.1, 4.2, and 4.3 also hold after the end of the update stage of the terminating epoch. This follows from the fact that the last communication stage of epoch $\lceil 1/\varepsilon \rceil - 2$ precedes the update stage of the terminating epoch and the fact that these last communication and update stages are no different from the communication and update stages of prior epochs (hence the same reasoning can be applied to obtain the result).

If after this last update, processor v is still a collector, meaning that v did not hear from all processors yet, according to the description of the algorithm, processor v will send a call-message to the processors whose pid is still in WAITING_v (by Lemma 4.3 and the update rule, it follows that list WAITING_v contains all processors that v did not hear from yet). Then all non-faulty processors w receive the call-message of v and then they respond to v. Then v receives these responses. Finally v updates list RUMORS_v accordingly: if a processor w responded to v's call-message (meaning that v now learns the rumor of w), then v adds (w, rumor_w) in RUMORS_v. If w did not respond to v's call-message, and (w, rumor_w) is not in RUMORS_v (it is possible for processor v to learn the rumor of w from some other processor v' that learned the rumor of w before processor w crashed), then v knows that w has crashed and adds $(w, \perp)$ in RUMORS_v.

Hence the last update that each non-faulty processor v performs on RUMORS_v maintains the validity that the list had from the previous epochs (guaranteed by the above three lemmas). Moreover, the size of RUMORS_v becomes equal to p and v either knows the rumor of each processor w, or it knows that v has crashed, as desired. □

Note from the above that the correctness of algorithm $\text{GOSSIP}_\varepsilon$ does not depend on whether the set of permutations Ψ satisfy the conditions of Corollary 3.5. The algorithm is correct for any set of permutations of $[p]$.

4.3. Analysis of algorithm $\text{GOSSIP}_\varepsilon$

Consider some set V_ℓ, $|V_\ell| \geqslant p^{1-\ell\varepsilon}$, of processors that are not idle at the beginning of epoch ℓ and do not crash by the end of epoch ℓ. Let $Q_\ell \subseteq V_\ell$ be such that $|Q_\ell| \geqslant |V_\ell|/7$ and the diameter of the subgraph induced by Q_ℓ is at most $31 \log p$. Q_ℓ exists because of Lemma 3.2 applied to graph G_ℓ and set V_ℓ.

For any processor v, let $\text{CALL}_v = \text{CALLING}_v \backslash \text{NEIGHB}_v$. Recall that the size of CALL is equal to $p^{(\ell+1)\varepsilon}$ (or less if list WAITING, or BUSY, is shorter than $p^{(\ell+1)\varepsilon}$) and the size of NEIGHB is at most $p^{(\ell+1)\varepsilon}$. We refer to the call-messages sent to the processors whose pids are in CALL as *progress-messages*. If processor v sends a progress-message to processor w, it will remove w from list WAITING_v (or BUSY_v) by the end of current stage. Let $d = (31 \log p + 1)\, p^{(\ell+1)\varepsilon}$. Note that $d \geqslant (31 \log p + 1) \cdot |\text{CALL}|$.

We begin the analysis of the gossip algorithm by proving a bound on the number of progress-messages sent under certain conditions.

Lemma 4.5. *The total number of progress-messages sent by processors in Q_ℓ from the beginning of epoch ℓ until the first processor in Q_ℓ will have its list* WAITING (*or list* BUSY) *empty, is at most $(d, |Q_\ell|)$-Surf(Ψ).*

Proof. Fix Q_ℓ and consider some permutation $\sigma \in S_p$ that satisfies the following property: "Consider $i < j \leqslant p$. Let τ_i (respectively τ_j) be the time step in epoch ℓ where some processor in Q_ℓ hears about $\sigma(i)$ (respectively $\sigma(j)$) the first time among the processors in Q_ℓ. Then $\tau_i \leqslant \tau_j$." (We note that it is not difficult to see that for a given Q_ℓ we can always find $\sigma \in S_p$ that satisfies the above property.) We consider only the subset $\Upsilon \subseteq \Psi$ containing permutations of indexes from set Q_ℓ. To show the lemma we prove that the number of

messages sent by processors from Q_ℓ is at most $(d, |\Upsilon|)$-Surf$(\Upsilon, \sigma) \leqslant (d, |Q_\ell|)$-Surf$(\Psi)$. Suppose that processor $v \in Q_\ell$ sends a progress-message to processor w. It follows from the diameter of Q_ℓ and the size of set CALL in epoch ℓ, that no processor $v' \in Q_\ell$ had sent a progress-message to w before $31 \log p$ phases, and consequently the position of processor w in permutation π_v is at most $d - |\text{CALL}| \leqslant d - p^{(\ell+1)\varepsilon}$ greater than the position of w in permutation $\pi_{v'}$.

For each processor $v \in Q_\ell$, let P_v contain all pairs (v, i) such that v sends a progress-message to processor $\pi_v(i)$ by itself during the epoch ℓ. We construct function h from the set $\bigcup_{v \in Q_\ell} P_v$ to the set of all d-lrm of set $\sigma^{-1} \circ \Psi$ and show that h is a one-to-one function. We run the construction independently for each processor $v \in Q_\ell$. If $\pi_v(k)$ is the first processor in the permutation π_v to whom v sends a progress-message at the beginning of epoch ℓ, we set $h(v, k) = 1$. Suppose that $(v, i) \in P_v$ and we have defined function h for all elements from P_v less than (v, i) in the lexicographic order. We define $h(v, i)$ as the first $j \leqslant i$ such that $(\sigma^{-1} \circ \pi_v)(j)$ is a d-lrm not assigned yet by h to any element in P_v.

Claim. *For every* $(v, i) \in P_v$, $h(v, i)$ *is well defined.*

We prove the claim. For the first element in P_v function h is well defined. For the first d elements in P_v it is also easy to show that h is well defined, since the first d elements in permutation π_v are d-lrms. Suppose h is well defined for all elements from P_v less than (v, i) and (v, i) is at least the $(d + 1)$st element in P_v. We show that $h(v, i)$ is also well defined. Suppose to the contrary, that there is no position $j \leqslant i$ such that $(\sigma^{-1} \circ \pi_v)(j)$ is a d-lrm and j is not assigned by h before the step of construction for $(v, i) \in P_v$. Let $j_1 < \cdots < j_d < i$ be the positions such that $(v, j_1), \ldots, (v, j_d) \in P_v$ and $(\sigma^{-1} \circ \pi_v)(h(j_1)), \ldots, (\sigma^{-1} \circ \pi_v)(h(j_d))$ are greater than $(\sigma^{-1} \circ \pi_v)(i)$. They exist from the fact, that $(\sigma^{-1} \circ \pi_v)(i)$ is not d-lrm and every "previous" d-lrms in π_v are assigned by L. Obviously processor $w = \pi_v(h(j_1))$ received a first progress-message at least $d/|\text{CALL}| \geqslant 31 \log p + 1$ phases before it received a progress-message from v. From the choice of σ, processor $w' = \pi_v(i)$ had received a progress-message from some other processor in Q'_ℓ at least $31 \log p + 1$ phases before w' received a progress-message from v. This contradicts the remark at the beginning of the proof of the lemma. This completes the proof of the claim.

The fact that h is a one-to-one function follows directly from the definition of h. It follows that the number of progress-messages sent by processors in Q_ℓ until the list WAITING (or list BUSY) of a processor in Q_ℓ is empty, is at most $(d, |\Upsilon|)$-Surf$(\Upsilon, \sigma) \leqslant (d, |Q_\ell|)$-Surf$(\Psi)$, as desired. □

We now define an invariant, that we call I_ℓ, for $\ell = 1, \ldots, \lceil 1/\varepsilon \rceil - 2$:

I_ℓ: There are at most $p^{1-\ell\varepsilon}$ non-faulty processors having status collector or informer in any step after the end of epoch ℓ.

Using Lemma 4.5 and Corollary 3.5 we show the following:

Lemma 4.6. *In any execution of algorithm* GOSSIP$_\varepsilon$, I_ℓ *holds for any epoch* $\ell = 1, \ldots, \lceil 1/\varepsilon \rceil - 2$.

Proof. For $p = 1$ it is obvious. Assume $p > 1$. We will use Lemma 3.2 and Corollary 3.5. Consider any epoch $\ell < \lceil 1/\varepsilon \rceil - 1$. Suppose to the contrary, that there is a subset V_ℓ

of non-faulty processors after the end of epoch ℓ such that each of them has status either collector or informer and $|V_\ell| > p^{1-\ell\varepsilon}$. Since G_ℓ satisfies $CCP(p, p-p^{1-\ell\varepsilon}, \varepsilon)$, there is a set $Q_\ell \subseteq V_\ell$ such that $|Q_\ell| \geqslant |V_\ell|/7 > p^{1-\ell\varepsilon}/7$ and the diameter of the subgraph induced by Q_ℓ is at most $31\log p$. Applying Lemma 4.5 and Corollary 3.5 to the set Q_ℓ, epoch ℓ, $t = p$, $q = |Q_\ell|$ and $d = 31p^{(\ell+1)\varepsilon}\log p$, we obtain that the total number of messages sent until some processor $v \in Q_\ell$ has list BUSY_v empty, is at most

$$\begin{aligned} &2 \cdot (31(\log p + 1)p^{(\ell+1)\varepsilon}, |Q_\ell|)\text{-Surf}(\Psi) + 31|Q_\ell|p^{(\ell+1)\varepsilon}\log p \\ &\leqslant 341|Q_\ell|p^{(\ell+1)\varepsilon}\log^2 p. \end{aligned}$$

More precisely, until some processor in Q_ℓ has status informer, the processors in Q_ℓ send at most $(31(\log p+1)p^{(\ell+1)\varepsilon}, |Q_\ell|)$-Surf$(\Psi)$ messages. Then, every processor in Q_ℓ has status informer after the processors in Q_ℓ send at most $31|Q_\ell|p^{(\ell+1)\varepsilon}\log p$ messages.

Finally, after the processors in Q_ℓ send at most $(31(\log p+1)p^{(\ell+1)\varepsilon}, |Q_\ell|)$-Surf$(\Psi)$ messages, some processor in $Q_\ell \subseteq V_\ell$ has its list BUSY empty.

Notice that since no processor in Q_ℓ has status idle in epoch ℓ, each of them sends in every phase of epoch ℓ at most $|\text{CALL}| \leqslant p^{(\ell+1)\varepsilon}$ progress-messages. Consequently the total number of phases in epoch ℓ until some of the processors in Q_ℓ has its list BUSY empty, is at most

$$\frac{341|Q_\ell|p^{(\ell+1)\varepsilon}\log^2 p}{|Q_\ell|p^{(\ell+1)\varepsilon}} = 341\log^2 p.$$

Recall that $\alpha\log^2 p > 341\log^2 p$. Hence if we consider the first $341\log^2 p$ phases of epoch ℓ, the above argument implies that there is at least one processor in V_ℓ that has status idle, which is a contradiction. Hence, I_ℓ holds for epoch ℓ. □

We now show the time and message complexity of algorithm $\text{GOSSIP}_\varepsilon$.

Theorem 4.7. *Algorithm* $\text{GOSSIP}_\varepsilon$ *solves the Gossip*(p, f) *problem with time complexity* $T = \text{O}(\log^2 p)$ *and message complexity* $M = \text{O}(p^{1+3\varepsilon})$.

Proof. First we show the bound on time. Observe that each update and communication stage takes O(1) time. Therefore each of the first $\lceil 1/\varepsilon\rceil - 2$ epochs takes $\text{O}(\log^2 p)$ time. The last epoch takes O(1) time. From this and the fact that ε is a constant, we have that the time complexity of the algorithm is in the worst case $\text{O}(\log^2 p)$.

We now show the bound on messages. From Lemma 4.6 we have that for every $1 \leqslant \ell < \lceil 1/\varepsilon\rceil - 2$, during epoch $\ell+1$ there are at most $p^{1-\ell\varepsilon}$ processors sending at most $2p^{(\ell+2)\varepsilon}$ messages in every communication stage. The remaining processors are either faulty (hence they do not send any messages) or have status idle—these processors only respond to call-messages and their total impact on the message complexity in epoch $\ell+1$ is at most as large as the others. Consequently the message complexity during epoch $\ell+1$ is at most $4(\alpha\log^2 p)\cdot(p^{1-\ell\varepsilon}p^{(\ell+2)\varepsilon}) \leqslant 4\alpha p^{1+2\varepsilon}\log^2 p \leqslant 4\alpha p^{1+3\varepsilon}$. After epoch $\lceil 1/\varepsilon\rceil - 2$ there are, per $\text{I}_{\lceil 1/\varepsilon\rceil-2}$, at most $p^{2\varepsilon}$ processors having list WAITING not empty. In epoch $\lceil 1/\varepsilon\rceil - 1$ each of these processors sends a message to at most p processors twice, hence the message complexity in this epoch is bounded by $2p\cdot p^{2\varepsilon}$. From the above and the fact that ε is a constant, we have that the message complexity of the algorithm is $\text{O}(p^{1+3\varepsilon})$. □

5. The Do-All algorithm based on gossip

We now put the gossip algorithm to use by constructing a new *Do-All* algorithm called algorithm DOALL_ε.

5.1. Description of algorithm DOALL_ε

The algorithm proceeds in a loop that is repeated until all the tasks are executed and all non-faulty processors are aware of this. A single iteration of the loop is called an *epoch*. Each epoch consists of $\beta \log p + 1$ *phases*, where $\beta > 0$ is a constant integer. We show that the algorithm is correct for any integer $\beta > 0$, but the complexity analysis of the algorithm depends on specific values of β that we show to exist. Each phase is divided into two *stages*, the *work* stage and the *gossip* stage. In the work stage processors perform tasks, and in the gossip stage processors execute an instance of the $\text{GOSSIP}_{\varepsilon/3}$ algorithm to exchange information regarding completed tasks and non-faulty processors (more details momentarily). Computation starts with epoch 1. We note that (unlike in algorithm $\text{GOSSIP}_\varepsilon$) the non-faulty processors may stop executing at different steps. Hence we need to argue about the termination decision that the processors must take. This is done in the paragraph "Termination decision".

The pseudocode of the algorithm is given in Fig. 2. The details are explained in the rest of this section. (Again we assume that every *if–then* has an implicit *else* containing no-ops as needed to ensure the synchrony of the system.)

Local knowledge. Each processor v maintains a list of tasks TASK_v it believes not to be done, and a list of processors PROC_v it believes to be non-faulty. Initially $\text{TASK}_v = \langle 1, \ldots, n \rangle$ and $\text{PROC}_v = \langle 1, \ldots, p \rangle$. The processor also has a Boolean variable $done_v$, that describes the knowledge of v regarding the completion of the tasks. Initially $done_v$ is set to false, and when processor v is assured that all tasks are completed $done_v$ is set to true.

Task allocation. Each processor v is equipped with a permutation π_v from a set Ψ of permutations on $[n]$.[5] We show that the algorithm is correct for any set of permutations on $[n]$, but its complexity analysis depends on specific set of permutations Ψ that we show to exist.

Initially TASK_v is permuted according to π_v and then processor v performs tasks according to the ordering of the tids in TASK_v. In the course of the computation, when processor v learns that task z is performed (either by performing the task itself or by obtaining this information from some other processor), it removes z from TASK_v while preserving the permutation order.

Work stage. For epoch ℓ, each work stage consists of $T_\ell = \lceil n + p \log^3 p/(p/2^l) \log p \rceil$ work *sub-stages*. In each sub-stage, each processor v performs a task according to TASK_v. Hence, in each work stage of a phase of epoch ℓ, processor v must perform the first T_ℓ tasks of TASK_v. However, if TASK_v becomes empty at a sub-stage prior to the T_ℓ^{th} sub-stage, then v performs no-ops in the remaining sub-stages (each no-op operation takes the same time as performing a task). Once TASK_v becomes empty, $done_v$ is set to true.

[5] This is distinct from the set of permutation on $[p]$ required by the gossip algorithm.

Initialization

$done_v = \mathsf{false}$;
$\text{TASK}_v = \langle \pi_v(1), \pi_v(2), \ldots, \pi_v(p) \rangle$;
$\text{PROC}_v = \langle 1, 2, \ldots, p \rangle$;

Do-All Algorithm

repeat *% iterate epochs until termination*

  repeat $\beta \log p + 1$ times *% iterating phases*

    repeat $T_\ell = \lceil \frac{n+p\log^3 p}{\frac{p}{2^\ell}\log p} \rceil$ times *% work stage of a phase begins*
      if TASK_v not empty **then**
        perform task whose id is first in TASK_v;
        remove task's id from TASK_v;
      elseif TASK_v empty and $done_v = \mathsf{false}$ **then**
        set $done_v$ to true;
    if TASK_v empty and $done_v = \mathsf{false}$ **then**
      set $done_v$ to true;

    run $\text{GOSSIP}_{\varepsilon/3}$ with $rumor_v$=(TASK_v, PROC_v,$done_v$); *% gossip stage of a phase begins*
    if $done_v = \mathsf{true}$ and $done_w = \mathsf{true}$ for all w received rumor from **then**
      TERMINATE;
    else
      update TASK_v and PROC_v; *% see paragraph Gossip stage*

Fig. 2. Algorithm DOALL_ε stated for processor v; $\pi_v(i)$ denotes the ith element of permutation π_v.

Gossip stage. Here processors execute algorithm $\text{GOSSIP}_{\varepsilon/3}$ using their local knowledge as the rumor, i.e., for processor v, $rumor_v = (\text{TASK}_v, \text{PROC}_v, done_v)$. At the end of the stage, each processor v updates its local knowledge based on the rumors it received. The *update rule* is as follows: (a) If v does not receive the rumor of processor w, then v learns that w has crashed (guaranteed by the correctness of $\text{GOSSIP}_{\varepsilon/3}$). In this case v removes w from PROC_v. (b) If v receives the rumor of processor w, then it compares TASK_v and PROC_v with TASK_w and PROC_w, respectively, and updates its lists accordingly—it removes the tasks that w knows are already completed and the processors that w knows that have crashed. Note that if TASK_v becomes empty after this update, variable $done_v$ remains false. It will be set to true in the next work stage. This is needed for the correctness of the algorithm (see Lemma 5.4).

Termination decision. We would like all non-faulty processors to learn that the tasks are done. Hence, it would not be sufficient for a processor to terminate once the value of its *done* variable is set to true. It has to be assured that all other non-faulty processors' *done* variables are set to true as well, and then terminate. This is achieved as follows: If processor v starts the gossip stage of a phase of epoch ℓ with $done_v = \mathsf{true}$, and all rumors it receives suggest that all other non-faulty processors know that all tasks are done (their *done* variables are set to true), then processor v terminates. If at least one processor's *done* variable is set

to false, then v continues to the next phase of epoch ℓ (or to the first phase of epoch $\ell+1$ if the previous phase was the last of epoch ℓ).

Remark 5.1. In the complexity analysis of the algorithm we first assume that $n \leqslant p^2$ and then we show how to extend the analysis for the case $n > p^2$. In order to do so, we assume that when $n > p^2$, before the algorithm DOALL_ε starts executing, the tasks are partitioned into $n' = p^2$ chunks, where each chunk contains at most $\lceil n/p^2 \rceil$ tasks. In this case it is understood that in the above description of the algorithm, n is actually n' and when we refer to a task we really mean a chunk of tasks.

5.2. *Correctness of algorithm* DOALL_ε

We show that the algorithm DOALL_ε solves the *Do-All*(n, p, f) problem correctly, meaning that the algorithm terminates with all tasks performed and all non-faulty processors are aware of this. Note that this is actually a stronger correctness condition than the one required in the definition of *Do-All*.

First we show that no non-faulty processor is removed from a processor's list of non-faulty processors.

Lemma 5.1. *In any execution of algorithm* DOALL_ε, *if processors* v *and* w *are non-faulty by the end of the gossip stage of phase* s *of epoch* ℓ, *then processor* w *is in* PROC_v.

Proof. Let v be a processor that is non-faulty by the end of the gossip stage of phase s of epoch ℓ. By the correctness of algorithm $\text{GOSSIP}_{\varepsilon/3}$ (called during the gossip stage), processor v receives the rumor of every non-faulty processor w and vice-versa. Since there are no restarts, v and w were alive in all prior phases of epochs $1, 2, \ldots, \ell$, and hence, v and w received each other rumors in all these phases as well. By the update rule it follows that processor v does not remove processor w from its processor list and vice versa. Hence w is in PROC_v and w is in PROC_v by the end of phase s, as desired. □

Next we show that no undone task is removed from a processor's list of undone tasks.

Lemma 5.2. *In any execution of algorithm* DOALL_ε, *if a task* z *is not in* TASK_v *of any processor* v *at the beginning of the first phase of epoch* ℓ, *then* z *has been performed in a phase of one of the epochs* $1, 2, \ldots, \ell-1$.

Proof. From the description of the algorithm we have that initially any task z is in TASK_v of a processor v. We proceed by induction on the number of epochs. At the beginning of the first phase of epoch 1, z is in TASK_v. If by the end of the first phase of epoch 1, z is not in TASK_v then by the update rule either (i) v performed task z during the work stage (hence the result follows), or (ii) during the gossip stage v received $rumor_w$ from processor w in which z was not in TASK_w. For the latter case, since this is the first epoch of the first phase, from the above and by the description of the algorithm we have that processor w performed task z during the work stage (hence the result follows). Continuing in this manner we have that if z is not in TASK_v at the beginning of the first phase of epoch 2, then z was performed in one of the phases of epoch 1.

Assuming that the thesis of the lemma holds for any epoch ℓ, we show that it also holds for epoch $\ell + 1$. Consider two cases:

Case 1: If z is not in TASK_v at the beginning of the first phase of epoch ℓ, then since no tid is ever added in TASK_v, z is not in TASK_v neither at the beginning of the first phase of epoch $\ell + 1$. By the inductive hypothesis, z was performed in one of the phases of epochs $1, \ldots, \ell - 1$.

Case 2: If z is in TASK_v at the beginning of the first phase of epoch ℓ but it is not in TASK_v at the beginning of the second phase of epoch ℓ, then by the update rule and the description of the algorithm it follows that either (i) v performed task z during the work stage of the second phase of epoch ℓ, or (ii) during the gossip stage of the second phase of epoch ℓ, v received $rumor_w$ from processor w in which z was not in TASK_w. For the latter case, from the above and the description of the algorithm we have that processor w performed task z during the work stage of the second phase of epoch ℓ or it learned that z was done in the gossip stage of the first phase of epoch ℓ. In either case, task z was performed. Continuing in this manner it follows that if z is not in TASK_v at the beginning of the first phase of epoch $\ell + 1$, then z was performed in one of the phases of epoch ℓ. □

Next we show that under certain conditions, local progress is guaranteed. First we introduce some notation. For processor v we denote by $\text{TASK}_v^{(\ell,s)}$ the list TASK_v at the beginning of phase s of epoch ℓ. Note that if s is the last phase—($\beta \log^2 p$)th phase—of epoch ℓ, then $\text{TASK}_v^{(\ell,s+1)} = \text{TASK}_v^{(\ell+1,1)}$, meaning that after phase s processor v enters the first phase of epoch $\ell + 1$.

Lemma 5.3. *In any execution of algorithm* DOALL_ε, *if processor* v *enters a work stage of a phase* s *of epoch* ℓ *with* $done_w = \mathsf{false}$ *and* TASK_v *not empty, then sizeof* $(\text{TASK}_v^{(\ell,s+1)}) <$ *sizeof* $(\text{TASK}_v^{(\ell,s)})$.

Proof. Let v be a processor that starts the work stage of phase s of epoch ℓ with $done_w = \mathsf{false}$. According to the description of the algorithm, the value of variable $done_v$ is initially false and it is set to true only when TASK_v becomes empty (it is possible however for TASK_v to be empty and $done_v$ to be still set on false, as $done_v$ is updated only in the work stage). Hence, if $done_w = \mathsf{false}$ and TASK_v is not empty at the beginning of the work stage of phase s of epoch ℓ there is at least one task identifier in $\text{TASK}_v^{(\ell,s)}$, and therefore v performs at least one task. From this and the fact that no tid is ever added in a processor's task list, we get that $sizeof(\text{TASK}_v^{(\ell,s+1)}) < sizeof(\text{TASK}_v^{(\ell,s)})$. □

We now show that when during a phase s of an epoch ℓ, a processor learns that all tasks are completed and it does not crash during this phase, then the algorithm is guaranteed to terminate by phase $s + 1$ of epoch ℓ; if s is the last phase of epoch ℓ, then the algorithm is guaranteed to terminate by the first phase of epoch $\ell + 1$. For simplicity of presentation, in the following lemma we assume that s is not the last phase of epoch ℓ.

Lemma 5.4. *In any execution of algorithm* DOALL_ε, *for any phase* s *of epoch* ℓ *and any processor* v, *if* $done_v$ *is set to* true *during phase* s *and* v *is non-faulty by the end of phase* s, *then the algorithm terminates by phase* $s + 1$ *of epoch* ℓ.

Proof. Consider phase s of epoch ℓ and processor v. According to the code of the algorithm, the value of variable $done_w$ is updated during the work stage of a phase (the value of the variable is not changed during the gossip stage). Hence, if the value of variable $done_w$ is changed during the phase s of epoch ℓ this happens before the start of the gossip stage. This means that TASK_v contained in $rumor_v$ in the execution of algorithm $\text{GOSSIP}_{\varepsilon/3}$ is empty. Since v does not fail during phase s, the correctness of algorithm $\text{GOSSIP}_{\varepsilon/3}$ guarantees that all non-faulty processors learn the rumor of v, and consequently they learn that all tasks are performed. This means that all non-faulty processors w start the gossip stage of phase $s+1$ of epoch ℓ with $done_w = \text{true}$ and all rumors they receive contain the variable *done* set to true.

The above, in conjunction with the termination guarantees of algorithm $\text{GOSSIP}_{\varepsilon/3}$, leads to the conclusion that all non-faulty processors terminate by phase $s+1$ (and hence the algorithm terminates by phase $s+1$ of epoch ℓ). □

Finally we show the correctness of algorithm DOALL_ε.

Theorem 5.5. *In any execution of algorithm* DOALL_ε, *the algorithm terminates with all tasks performed and all non-faulty processors being aware of this.*

Proof. By Lemma 5.1, no non-faulty processor leaves the computation, and by our model at least one processor does not crash ($f < p$). Also from Lemma 5.2 we have that no undone task is removed from the computation. From the code of the algorithm we get that a processor continues performing tasks until its TASK list becomes empty and by Lemma 5.3 we have that local progress is guaranteed. The above, in conjunction with the correctness of algorithm $\text{GOSSIP}_{\varepsilon/3}$, leads to the conclusion that there exist a phase s of an epoch ℓ and a processor v so that during phase s processor v sets $done_v$ to true, all tasks are indeed performed and v survives phase s. By Lemma 5.4 the algorithm terminates by phase $s+1$ of epoch ℓ (or by the first phase of epoch $\ell+1$ if s is the last phase of epoch ℓ). Now, from the definition of T_ℓ it follows that the algorithm terminates after at most $O(\log p)$ epochs: consider epoch $\log p$; $T_{\log p} = \lceil (n + p\log^3 p)/\log p \rceil = \lceil n/\log p + p\log^2 p \rceil$. Recall that each epoch consists of $\beta \log p + 1$ phases. Say that $\beta = 1$. Then, when a processor reaches epoch $\log p$, it can perform all n tasks in this epoch. Hence, all tasks that are not done until epoch $\log p - 1$ are guaranteed to be performed by the end of epoch $\log p$ and all non-faulty processors will know that all tasks have been performed. □

Note from the above that the correctness of algorithm DOALL_ε does not depend on the set of permutations that processors use to select what tasks to do next. The algorithm works correctly for any set of permutations on $[n]$. It also works for any integer $\beta > 0$.

5.3. Analysis of algorithm DOALL_ε

We now derive the work and message complexities for algorithm DOALL_ε. Our analysis is based on the following terminology. For the purpose of analysis, we number globally all phases in the execution by positive integers starting from 1. Consider phase i which belongs to some epoch ℓ. For a given failure pattern F, let $V_i(F)$ denote the set of processors that

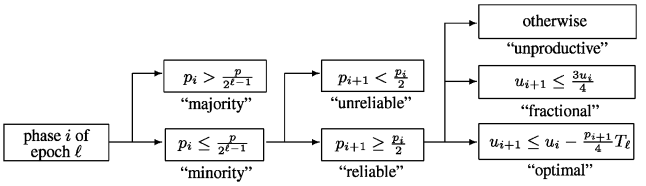

Fig. 3. Classification of a phase i of epoch ℓ; the failure pattern F is implied.

are non-faulty at the beginning of phase i. Let $p_i(F) = |V_i(F)|$. Let $U_i(F)$ denote the set of tasks z such that z is in some list TASK_v, for some $v \in V_i(F)$, at the beginning of phase i. Let $u_i(F) = |U_i(F)|$.

Now we classify the possibilities for phase i as follows. If at the beginning of phase i, $p_i(F) > p/2^{\ell-1}$, we say that phase i is a *majority* phase. Otherwise, phase i is a *minority* phase. If phase i is a minority phase and at the end of i the number of surviving processors is less than $p_i(F)/2$, i.e., $p_{i+1}(F) < p_i(F)/2$, we say that i is an *unreliable* minority phase. If $p_{i+1}(F) \geqslant p_i(F)/2$, we say that i is a *reliable* minority phase. If phase i is a reliable minority phase and $u_{i+1}(F) \leqslant u_i(F) - \frac{1}{4}p_{i+1}(F)T_\ell$, then we say that i is an *optimal* reliable minority phase (the task allocation is optimal; the same task is performed only by a constant number of processors on average). If $u_{i+1}(F) \leqslant \frac{3}{4}u_i(F)$, then i is a *fractional* reliable minority phase (a fraction of the undone tasks is performed). Otherwise we say that i is an *unproductive* reliable minority phase (not much progress is obtained). The classification possibilities for phase i of epoch ℓ are depicted in Fig. 3.

Our goal is to choose a set Ψ of permutations and a constant $\beta > 0$ such that for any failure pattern there will be no unproductive and no majority phases. To do this we analyze sets of random permutations, prove certain properties of our algorithm for such sets (in Lemmas 5.6 and 5.7), and finally use the probabilistic method to obtain an existential deterministic solution.

We now give the intuition why the phases, with high probability, are neither majority nor minority reliable unproductive. First, in either of such cases, the number of processors crashed during the phase is at most half of all operational processors during the phase. Consider only those majorities of processors that survive the phase and the tasks performed by them. If there are a lot of processors, then all tasks will be performed if the phase is a majority phase, or at least $\min\{u_i(F), |Q|T_\ell\}/4$ yet unperformed tasks are performed by the processors if the phase is a minority reliable unproductive phase, all with high probability. Hence we can derandomize the choice of suitable set of permutations such that for any failure pattern there are neither majority nor minority reliable unproductive phases.

Note that these observations suggest an approach to a failure-sensitive algorithm. However, our algorithm is not optimal (in asymptotic sense) with respect to failure-sensitivity, so we propose a modified approach to this problem in Section 5.4.

Lemma 5.6. *Consider a fixed nonempty subset Q of processors in phase i of epoch ℓ of algorithm* DOALL_ε. *Then the probability of event "for every failure pattern F such that $V_{i+1}(F) \supseteq Q$ and $u_i(F) > 0$, the following inequality holds $u_i(F) - u_{i+1}(F) \geqslant \min\{u_i(F), |Q|T_\ell\}/4$" is at least $1 - \mathrm{e}^{-|Q|T_\ell/8}$.*

Proof. Let $c = \min\{u_i(F), |Q|T_\ell\}/4$. Let F be a failure pattern such that $V_{i+1}(F) \supseteq Q$ and $u_i(F) > 0$. Let $S_i(F)$ be the set of tasks z such that z is in every list TASK_v for $v \in Q$, at the beginning of phase i. Let $s_i(F) = |S_i(F)|$. Note that $S_i(F) \subseteq U_i(F)$, and that $S_i(F)$ describes some properties of set Q, while $U_i(F)$ describes some properties of set $V_i(F) \supseteq Q$.

Consider the following two cases:

Case 1: $s_i(F) \leqslant u_i(F) - c$. Then after the gossip stage of phase i we obtain the required inequality with probability 1.

Case 2: $s_i(F) > u_i(F) - c$. We focus on the work stage of phase i. Consider a conceptual process in which the processors in Q perform tasks sequentially, the next processor takes over when the previous one has performed all its T_ℓ steps during work stage of phase i. This process takes $|Q|T_\ell$ steps to be completed. Let $U_i^{(k)}(F)$ denote the set of tasks z such that: z is in some list TASK_v, for some $v \in Q$, at the beginning of phase i and z has not been performed during the first k steps of the process, by any processor. Let $u_i^{(k)}(F) = |U_i^{(k)}(F)|$. Define the random variables X_k, for $1 \leqslant k \leqslant |Q|T_\ell$, as follows:

$$X_k = \begin{cases} 1 & \text{if either } u_i(F) - u_i^{(k)}(F) \geqslant c \text{ or } u_i^{(k)}(F) \neq u_i^{(k-1)}(F), \\ 0 & \text{otherwise.} \end{cases}$$

Suppose some processor $v \in Q$ is to perform the kth step. If $u_i(F) - u_i^{(k)}(F) < c$ then we also have the following:

$$s_i(F) - \left(u_i(F) - u_i^{(k)}(F)\right) > s_i(F) - c \geqslant u_i(F)/2 \geqslant \textit{sizeof}(\text{TASK}_v)/2,$$

where TASK_v is taken at the beginning of phase i, because $3c \leqslant 3u_i(F)/4 \leqslant s_i(F)$. Thus at least a half of the tasks in TASK_v, taken at the beginning of phase i, have not been performed yet, and so $\Pr[X_k = 1] \geqslant \frac{1}{2}$.

We need to estimate the probability $\Pr[\sum X_k \geqslant c]$, where the summation is over all $|Q|T_\ell$ steps of all the processors in Q in the considered process. Consider a sequence $\langle Y_k \rangle$ of independent Bernoulli trials, with $\Pr[Y_k = 1] = \frac{1}{2}$. Then the sequence $\langle X_k \rangle$ statistically dominates the sequence $\langle Y_k \rangle$, in the sense that

$$\Pr\left[\sum X_k \geqslant d\right] \geqslant \Pr\left[\sum Y_k \geqslant d\right]$$

for any $d > 0$. Notice that $\mathbb{E}[\sum Y_k] = |Q|T_\ell/2$ and $c \leqslant \mathbb{E}[\sum Y_k]/2$, hence we can apply Chernoff bound to obtain

$$\Pr\left[\sum Y_k \geqslant c\right] \geqslant 1 - \Pr\left[\sum Y_k < \frac{1}{2}\mathbb{E}\left[\sum Y_k\right]\right] \geqslant 1 - \mathrm{e}^{-|Q|T_\ell/8}.$$

Hence the number of tasks in $U_i(F)$, for any F such that $V_{i+1}(F) \supseteq Q$, performed by processors from Q during work stage of phase i is at least c with probability $1 - \mathrm{e}^{-|Q|T_\ell/8}$. □

Lemma 5.7. *Assume $n \leqslant p^2$ and $p \geqslant 2^8$. There exists a constant integer $\beta > 0$ such that for every phase i which is in some epoch $\ell > 1$, if there is a task unperformed by the beginning*

of phase i then

(a) *the probability that phase i is a majority phase under some failure pattern F is at most* $e^{-p \log p}$, *and*

(b) *the probability that phase i is a minority reliable unproductive phase under some failure pattern F is at most* $e^{-T_\ell/16}$.

Proof. We first prove clause (a). Assume that phase i belongs to epoch ℓ, for some $\ell > 1$. First we group failure patterns F such that phase i is a majority phase in F, according to the following equivalence relation: failure patterns F_1 and F_2 are in the same class iff $V_{i+1}(F_1) = V_{i+1}(F_2)$. Every such equivalence class is represented by some set of processors Q of size greater than $p/2^{\ell-1}$, such that for every failure pattern F in this class we have $V_{i+1}(F) = Q$. In the following claim we define conditions for β for satisfying clause (a).

Claim. *For constant* $\beta = 9$ *and under any failure pattern F in the class represented by Q, where* $|Q| > p/2^{\ell-1}$, *all tasks were performed by the end of epoch* $\ell - 1$ *with probability at least* $1 - e^{-p \log p - p}$.

We prove the claim. Let F be any failure pattern from a class represented by Q. Consider all steps taken by processors in Q during phase j of epoch $\ell - 1$. By Lemma 5.6, since $V_{j+1}(F) \supseteq Q$, we have that the probability of event "if $u_j(F) > 0$ then $u_j(F) - u_{j+1}(F) \geqslant \min\{u_j(F), |Q|T_{\ell-1}\}/4$", is at least $1 - 1/e^{|Q|T_{\ell-1}/8}$. If the above condition is satisfied we call phase j productive (for consistency with the names optimal and fractional; the difference is that these names are used only for minority phases—now we use it according to the progress made by processors in Q). Phase j might be productive with probability at least $1 - 1/e^{|Q|T_{\ell-1}/8}$. Since the total number of tasks is n, we have that the number of productive phases during epoch $\ell - 1$ sufficient to perform all tasks using only processors in Q is either at most

$$\frac{n}{|Q|T_{\ell-1}/4} \leqslant \frac{n}{n/(4 \log p)} = 4 \log p$$

or, since $n \leqslant p^2$, at most

$$\log_{4/3} n = 5 \log p.$$

Therefore there are a total of $9 \log p$ productive phases, which are sufficient to perform all the tasks. Furthermore, every phase in epoch $\ell - 1$ is productive. Hence, all tasks are performed by processors in Q during $\beta \log p$ phases, for constant $\beta = 9$, of epoch $\ell - 1$ with probability at least

$$1 - 9 \log p \cdot e^{-|Q|T_{\ell-1}/8} \geqslant 1 - e^{\ln 9 + \ln \log p - (p \log^2 p)/4} \geqslant 1 - e^{-p \log p - p},$$

since $p \geqslant 8$. Consequently all processors terminate by the end of phase $\beta \log p + 1$ with probability at least $1 - e^{-p \log p - p}$. This follows by the correctness of the gossip algorithm and the argument of Lemma 5.4, since epoch $\ell - 1$ lasts $\beta \log p + 1$ phases and processors in Q are non-faulty at the beginning of epoch ℓ. This completes the proof of the claim.

There are at most 2^p possible sets Q of processors, hence by the claim the probability that phase i is a majority phase is at most

$$2^p \cdot \mathrm{e}^{-p\log p - p} \leqslant \mathrm{e}^{-p\log p},$$

which proves clause (a) for phase i.

Now we prove clause (b) for phase i. Assume that phase i in epoch ℓ is a minority reliable phase. Similarly as above, we partition all failure patterns F according to the following equivalence relation: failure patterns F_1 and F_2 are in the same class if there is a set Q such that $H = V_{i+1}(F_1) = V_{i+1}(F_2)$. Set Q is a representative of a class. By Lemma 5.6 applied to phase i and set Q we obtain that the probability that phase i is unproductive for every failure pattern F such that $V_{i+1}(F) = Q$ is $\mathrm{e}^{-|Q|T_\ell/8}$. Hence the probability that for every failure pattern F phase i is a minority reliable unproductive phase is at most

$$\sum_{x=1}^{p/2^{\ell-1}} \binom{p}{x} \cdot \mathrm{e}^{-xT_\ell/8} \leqslant \sum_{x=1}^{p/2^{\ell-1}} 2^{x\log p} \cdot \mathrm{e}^{-xT_\ell/8} \leqslant \sum_{x=1}^{p/2^{\ell-1}} \mathrm{e}^{x\log p - xT_\ell/8}$$
$$\leqslant \mathrm{e}^{\log p - T_\ell/8} \cdot \frac{1}{1-\mathrm{e}^{\log p - T_\ell/8}} \leqslant \mathrm{e}^{-T_\ell/16},$$

(since $p \geqslant 2^8$), showing clause (b) for phase i. □

Recall that epoch ℓ consists of $\beta\log p + 1$ phases for some $\beta > 0$ and that $T_\ell = \lceil n + p\log^3 p/(p/2^\ell)\log p\rceil$. Also by the correctness proof of algorithm DOALL$_\varepsilon$ (Theorem 5.5), the algorithm terminates in $O(\log p)$ epochs, hence, the algorithm terminates in $O(\log^2 p)$ phases. Let g_ℓ be the number of steps that each gossip stage takes in epoch ℓ, i.e., $g_\ell = \Theta(\log^2 p)$.

We now show the work and message complexity of algorithm DOALL$_\varepsilon$.

Theorem 5.8. *There is a set of permutations Ψ and a constant integer $\beta > 0$ (e.g., $\beta = 9$) such that algorithm* DOALL$_\varepsilon$, *using permutations from Ψ, solves the Do-All(n, p, f) problem with work $W = O(n + p\log^3 p)$ and message complexity $M = O(p^{1+2\varepsilon})$.*

Proof. We show that for any execution $\mathcal{E}$ of algorithm DOALL$_\varepsilon$ that solves the $Do - All$ (n, p, f) problem under a failure pattern F, there exists a set of permutations Ψ and an integer $\beta > 0$ so that the complexity bounds are as desired. Let β be from Lemma 5.7. We consider two cases:

Case 1: $n \leqslant p^2$. Consider phase i of epoch ℓ of algorithm DOALL$_\varepsilon$ for randomly chosen set of permutations Ψ. We reason about the probability of phase i belonging to one of the classes illustrated in Fig. 3, and about the work that phase i contributes to the total work incurred in the execution, depending on its classification. From Lemma 5.7(a) we get that phase i may be a majority phase under any failure pattern F with probability $\mathrm{e}^{-\Omega(p\log p)}$ which is a very small probability. More precisely, the probability that for some failure pattern F and set of permutations Ψ, in execution $\mathcal{E}$ obtained for F and Ψ there is a majority phase, is $O(\log^2 p \cdot \mathrm{e}^{-p\log p}) = \mathrm{e}^{-\Omega(p\log p)}$ (recall that the execution has $O(\log^2 p)$ phases), and consequently using the probabilistic method argument we obtain that for almost any set of permutations Ψ there is no execution in which there is a majority phase.

Therefore, we focus on minority phases that occur with high probability (per Lemma 5.7(a)). We cannot say anything about the probability of a minority phase to be reliable or unreliable, since this depends on F. Note however, that by definition, we cannot have more than $O(\log p)$ unreliable minority phases in any execution $\mathcal{E}$ (at least one processor must remain operational). Moreover, the work incurred in an unreliable minority phase i of an epoch ℓ in any execution $\mathcal{E}$ is bounded by

$$\begin{aligned} O\left(p_i(F)\cdot(T_\ell+g_\ell)\right) &= O\left(\frac{p}{2^{\ell-1}}\cdot\left(\frac{n+p\log^3 p}{p/2^\ell\log p}+\log^2 p\right)\right)\\ &= O\left(\frac{n}{\log p}+p\log^2 p\right). \end{aligned}$$

Thus, the total work incurred by all unreliable minority phases in any execution $\mathcal{E}$ is $O(n+p\log^3 p)$.

From Lemmas 5.6 and 5.7(b) we get that a reliable minority phase may be fractional or optimal for some failure pattern F with high probability $1-e^{-T_\ell/16}$, whereas it may be unproductive for some failure pattern F with very small probability $e^{-T_\ell/16}\leqslant e^{-(\log^2 p)/16}$. Using a similar argument as for majority phases, we get that for almost all sets of permutations Ψ (probability $1-O(\log^2 p\cdot e^{-T_\ell/16})\geqslant 1-e^{-\Omega(T_\ell)}$) and for every failure pattern F, and hence for every execution $\mathcal{E}$, there is no minority reliable unproductive phase. The work incurred by a fractional phase i of an epoch ℓ, under any failure pattern F and execution $\mathcal{E}$, is bounded by $O(p_i(F)\cdot(T_\ell+g_\ell)) = O(n/\log p+p\log^2 p)$. Also note that by definition, there can be at most $O(\log_{4/3} n)$ $(=O(\log p)$ since $n\leqslant p^2)$ fractional phases in any execution $\mathcal{E}$ and hence, the total work incurred by all fractional reliable minority phases in any execution $\mathcal{E}$ is $O(n+p\log^3 p)$. We now consider the optimal reliable minority phases under any failure pattern F and execution $\mathcal{E}$. Here we have an optimal allocation of tasks to processors in $V_i(F)$. By definition of optimality, in average one task in $U_i(F)\backslash U_{i+1}(F)$ is performed by at most *four* processors from $V_{i+1}(F)$, and by definition of reliability, by at most *eight* processors in $V_i(F)$. Therefore, in optimal phases, each unit of work spent on performing a task results to a unique task completion (with a constant overhead), for any execution $\mathcal{E}$. It therefore follows that the work incurred in all optimal reliable minority phases is bounded by $O(n)$ in any execution $\mathcal{E}$.

Therefore, from the above we conclude that when $n\leqslant p^2$, for random set of permutations Ψ the work complexity of algorithm DoAll$_\varepsilon$ executed on such set Ψ and under any failure pattern F is $W = O(n+p\log^3 p)$ with probability $1-e^{-\Omega(p\log p)}-e^{-\Omega(T_\ell)} = 1-e^{-\Omega(T_\ell)}$ (the probability appears only from the analysis of majority and unproductive reliable minority phases). Consequently such set Ψ exists. Also, from Lemma 5.7 and the above discussion, $\beta>0$ (e.g., $\beta=9$) exists. Finally, the bound on messages using selected set Ψ and constant β is obtained as follows: there are $O(\log^2 p)$ executions of gossip stages. Each gossip stage requires $O(p^{1+\varepsilon})$ messages (message complexity of one instance of Gossip$_{\varepsilon/3}$). Thus, $M = O(p^{1+\varepsilon}\log^2 p) = O(p^{1+2\varepsilon})$.

Case 2: $n > p^2$. In this case, the tasks are partitioned into $n' = p^2$ chunks, where each chunk contains at most $\lceil n/p^2\rceil$ tasks (see Remark 5.1). Using the result of Case 1 and selected set Ψ and constant β, we get that $W = O(n'+p\log^3 p)\cdot\Theta(n/p^2) = O(p^2\cdot n/p^2+n/p^2\cdot p\log^3 p) = O(n)$. The message complexity is derived with the same way as in Case 1. □

5.4. Sensitivity training and failure-sensitive analysis

We note that the complexity bounds we obtained in the previous section do not show how the bounds depend on f, the maximum number of crashes. In fact it is possible to subject the algorithm to "failure-sensitivity-training" and obtain better results. To do so we slightly modify algorithm DOALL$_{\varepsilon/2}$ and obtain an algorithm we call DOALL$'_{\varepsilon}$. We first describe and analyze the modified version of algorithm GOSSIP$_{\varepsilon}$, called GOSSIP$'_{\varepsilon}$, which algorithm DOALL$'_{\varepsilon}$ uses as a building block (in a similar manner that algorithm DOALL$_{\varepsilon}$ uses algorithm GOSSIP$_{\varepsilon}$) to solve the *Do-All* problem. Then we present algorithm DOALL$'_{\varepsilon}$ and its analysis.

5.4.1. Algorithm GOSSIP$'_{\varepsilon}$

Algorithm GOSSIP$'_{\varepsilon}$ is a modified version of algorithm GOSSIP$_{\varepsilon}$. In particular, algorithm GOSSIP$'_{\varepsilon}$ contains a new epoch, called epoch 0. Epochs $1, \ldots, \lceil 1/\varepsilon \rceil - 1$ are the same epochs as in algorithm GOSSIP$_{\varepsilon}$. Assume for simplicity of presentation that $p/\log^2 p$ is an even integer. Epoch 0 is similar to the epoch 1 of algorithm GOSSIP$_{\varepsilon}$, except from the following:

- Epoch 0 contains $\alpha' \log^2 p$ phases, for some positive constant α', possibly different than α from algorithm GOSSIP$_{\varepsilon}$;
- The communication graph G_0 used in epoch 0 is defined as follows: let V' be the set consisting of arbitrarily chosen $2p/\log^2 p$ processors from V, where V denotes the set of all processors ($V = [p]$); G_0 is a graph on the set of nodes V' satisfying PROPERTY $\mathcal{R}(|V'|, |V'|/2)$.
- The processors in V' perform the normal phase of an epoch of algorithm GOSSIP$_{\varepsilon}$.
- To every processor in V' we attach one permutation from the set Ψ consisting of $2p/\log^2 p$ permutations from set S_p; we show in the analysis that suitable set Ψ exists.
- For every processor $v \in V'$, the size of set CALLING$_v \setminus$ NEIGHB$_v$ is equal 1.
- The processors that are not in V' perform a different code of the phase: they begin with a new status answer and do not change it by the end of epoch 0; if during epoch 0 processor $v \notin V'$ receives a message from a processor of status collector or informer, it answers to this processor in the same communication stage.
- If at the end of epoch 0, for processor v, $sizeof$(RUMORS$_v$) $= p$, then v sets its status to idle and removes its pid from list BUSY$_v$, otherwise v sets its status to collector.

Remark 5.2. Note that each processor that sets its status to idle at the end of epoch 0 might have its list BUSY not empty, as opposed to the processors that become idle after epoch greater than 0, where their list BUSY is empty. However, this does not affect the correctness of the epochs of number greater than 0: list BUSY is used by each processor to decide the subset of the processors it sends a call-message at each step of the computation (when the processor has status informer) and once it becomes empty, the processor sets it status to idle. According to the code of the algorithm, processors that are idle do not send call messages (they only respond to such messages). Therefore, the processors that become idle by the end of epoch 0 no longer use their list BUSY (whether is empty or not). However, it is important to notice that they remove their pid from their list BUSY so that when their local information is propagated to other processors (via responses to call messages), the other processors get to know that these processors are no longer collectors.

We now prove the complexity of algorithm GOSSIP$'_\varepsilon$.

Theorem 5.9. *There exist constant α' and set Ψ such that algorithm* GOSSIP$'_\varepsilon$, *using set Ψ, solves the Gossip(p, f) problem with time complexity $T = O(\log^2 p)$ and message complexity $M = O(p)$ when $f \leqslant p/\log^2 p$, and with $T = O(\log^2 p)$ and $M = O(p^{1+3\varepsilon})$ otherwise.*

Proof. First we consider the case where there are at most $p/\log^2 p$ failures by the end of epoch 0. Let $Q' \subseteq V'$ be a set of processors such that $|Q'| \geqslant |V'|/2 \geqslant p/\log^2 p$. By PROPERTY $\mathcal{R}(|V'|, |V'|/2)$ there exists $Q \subseteq Q'$ such that $|Q| \geqslant |Q'|/7$ and the diameter of graph G_Q is at most $31 \log p$. Consider all failure patterns F such that every processor in Q' is not crashed by the end of epoch 0, and choose Ψ randomly. We may look at the process of collecting rumors by processors in Q (when every processor in Q works as a collector) as a performing tasks: if a rumor of processor w (or information that processor w is crashed), for every processor w, is known by some processor in Q then we say that task w is performed. We partition every execution into consecutive *blocks*, each containing $31 \log p$ consecutive phases. Notice that during each block all processors in Q exchange information between themselves, by definition of Q. We may use Lemma 5.6 to bound progress: the probability that "for every considered failure pattern F (such that all processors in Q are not crashed at the end of epoch 0) after every consecutive block in epoch 0 the number of rumors unknown by processors in Q decreases either by $(3/4)|Q| \log p$ or by factor $3/4$" is $1 - e^{-\Omega(|Q| \log p)}$. Consequently, for every considered failure pattern F, $O(p/|Q| \log p + \log_{3/4} p) = O(\log p)$ number of blocks are sufficient to collect all rumors by processors in Q, with probability at least $1 - \log p \cdot e^{-\Omega(|Q| \log p)} \geqslant 1 - e^{-\Omega(|Q| \log p)}$. Using the probabilistic method we choose one such Ψ, which additionally satisfies the thesis of Theorem 3.4 (to assure that Ψ is good also for the other cases in this proof) and constant α' follows from the fact that $O(\log p)$ blocks, each of $31 \log p$ phases, suffice to collect all rumors by processors in Q for every failure pattern F.

The process in which processors in Q, acting as informer, inform all other processors about collected rumors and the status of all processors, is similar to the process of collecting, and do not influence the asymptotic complexity. In this case performing task w, for every processor w, is defined as informing processor w by some processor in Q.

Since the communication graph G has constant degree and in every phase the size of set $\text{CALLING}_v \setminus \text{NEIGHB}_v$ is equal to 1, the number of messages sent in every phase is $O(|V'|) = O(p/\log^2 p)$, which, in view of the number $O(\log^2 p)$ of phases in epoch 0, gives message complexity $O(p)$ in epoch 0.

Consider the case where at the end of epoch 0 there are more than $p/\log^2 p$ faulty processors. In this case there may be some processor $v \in V$ which has its list RUMORS not filled at the end of epoch 0 (if not then all non-faulty processors become idle at the end of epoch 0 and we are done). It follows that all such processors start executing epoch 1 of algorithm GOSSIP$'_\varepsilon$ which is the same as in algorithm GOSSIP$_\varepsilon$.

Using the same argument as in the proof of Theorem 4.7 and by the fact that Ψ was chosen to satisfy the thesis of Theorem 3.4, we obtain that the message complexity during execution of GOSSIP$'_\varepsilon$ is $O(p^{1+2\varepsilon} \log^3 p) = O(p^{1+3\varepsilon})$, which together with $O(p)$ messages sent in epoch 0 yields the thesis of the theorem, with respect to message complexity. The

time complexity yields from the fact that epoch 0 runs $O(\log^2 p)$ phases, and the remaining epochs run also for $O(\log^2 p)$ phases. □

5.4.2. Algorithm $\text{DOALL}'_\varepsilon$

Algorithm $\text{DOALL}'_\varepsilon$ is a modified version of algorithm $\text{DOALL}_{\varepsilon/2}$. In particular, algorithm $\text{DOALL}'_\varepsilon$ contains two new epochs, called epoch -1 and epoch 0. Epochs $1, \ldots, \log p$ are the same epochs as in algorithm $\text{DOALL}_{\varepsilon/2}$.

Epoch -1 of algorithm $\text{DOALL}'_\varepsilon$ uses the check-pointing algorithm from [9], where the check-pointing and the synchronization procedures are taken from [11]. We refer to the algorithm used in epoch -1 as algorithm DGMY. The goal of using this algorithm in epoch -1 is to solve *Do-All* with work $O(n+p(f+1))$ and communication $O(fp^\varepsilon+p\ \min\{f+1, \log p\})$ if the number of failures is small, mainly concerning the case $f \leqslant \log^3 p$. Hence, in epoch -1, we execute DGMY *only until* step $a \cdot (n/p + \log^3 p)$, for some constant a such that the early-stopping condition of DGMY holds for every $f \leqslant \log^3 p$.

Epoch 0 of algorithm $\text{DOALL}'_\varepsilon$ is similar to an epoch of algorithm $\text{DOALL}_{\varepsilon/2}$, except that instead of algorithm $\text{GOSSIP}_{\varepsilon/3}$, we use algorithm $\text{GOSSIP}'_{\varepsilon/6}$ in each gossip stage of every phase of epoch 0. Each gossip stage lasts $g_0 = \alpha' \log^2 p$ steps, for a fixed constant α' which depends on algorithm $\text{GOSSIP}'_{\varepsilon/6}$.

We now show the work and message complexity of algorithm $\text{DOALL}'_\varepsilon$.

Theorem 5.10. *There exists a set of permutations Ψ and a constant integer $\beta > 0$ such that algorithm* $\text{DOALL}'_\varepsilon$ *solves the Do-All(n, p, f) problem with work $W = O(n + p \cdot \min\{f + 1, \log^3 p\})$ and message complexity $M = O(fp^\varepsilon + p\ \min\{f + 1, \log p\})$.*

Proof. We consider three cases:

Case 1: If the number of failures f during the execution of DGMY (recall that we execute the algorithm up to step $a \cdot (n/p + \log^3 p)$) is not greater than $\log^3 p$ then by the early stopping property of algorithm DGMY, all non-faulty processors terminate by the end of this execution of DGMY. The work performed by the algorithm is $O(n + (f + 1)p)$ and the message complexity is $O(fp^\varepsilon + p\ \min\{f + 1, \log p\})$. This follows from the results in [9] and [11].

Case 2: If the number of failures f during the execution of DGMY is greater than $\log^3 p$ and some processor terminates in epoch -1, then by the correctness of algorithm DGMY all tasks are performed. Hence, as in the previous case, work performed by the algorithm is $O(n + (f + 1)p)$ and the message complexity is $O(fp^\varepsilon + p\ \min\{f + 1, \log p\})$.

Case 3: If the number of failures f during the execution of DGMY is greater than $\log^3 p$ and no processor terminates during the execution of DGMY, then every non-faulty processor, unlike the previous two cases, starts executing epoch 0 of $\text{DOALL}'_\varepsilon$, each at the same time. The work during the execution of DGMY is $O(n+p\log^3 p) = O(n+p\cdot\min\{f+1, \log^3 p\})$ and the message complexity is $O(f'p^\varepsilon + p\ \min\{f' + 1, \log p\})$, where $f' \leqslant f$ is the number of crashes occurred during epoch -1. We now analyze the work and communication complexity of the remaining epochs.

The analysis of the remaining epochs, starting from epoch 0, is done similarly as in Theorem 5.8. The only difference in the analysis is that we use one more epoch (epoch 0), in which the message complexity of every gossip stage is $O(p)$, if $f \leqslant p/\log^2 p$

(per Theorem 5.9). Notice that the total number of phases is still $O(\log^2 p)$, as used in the proof of Theorem 5.8 (but the constant may differ from the original). Hence the choice of set Ψ is the same as in the proof of Theorem 5.8, as well as the conditions for an integer constant $\beta > 0$, where $\beta \log p + 1$ is the number of phases in one epoch (only the constants hidden in asymptotic notation may differ, and this may increase the constant β with respect to the original one). The analysis for the general case where $f < p$ is the same as in the proof of Theorem 5.8. Therefore, we only consider failure patterns F such that $f \leqslant p/\log^2 p$. We have $|V_i(F)| \geqslant p - p/\log^2 p$ for every phase i in epoch 0, and consequently the number of phases in epoch 0 sufficient to perform all the tasks, which (by the proof of Theorem 5.8 means performing work $O(n + p \log^3 p)$) is

$$O\left(\frac{n + p\log^3 p}{T_0 \cdot (p - p/\log^2 p)}\right) = O\left(\frac{n + p\log^3 p}{[n/(p\log p) + \log^2 p] \cdot (p - p/\log^2 p)}\right) = O(\log p).$$

The constant integer $\beta > 0$ must satisfy the conditions imposed to it in the proof of Theorem 5.8. In addition, β must be such, that the constant hidden in the above $O(\log p)$ notation must be less than β. If we choose a β that satisfies all the above-mentioned conditions, then we have that for every failure pattern F, and hence (since Ψ is fixed) for every execution $\mathcal{E}$ such that $f \leqslant p/\log^2 p$, algorithm $\text{DOALL}'_\varepsilon$ terminates by the end of epoch 0. Also, by the complexity of algorithm $\text{GOSSIP}'_{\varepsilon/6}$ shown in Theorem 5.9, we have that the total number of messages sent is $O(p \cdot \log p) = O(p \min\{f + 1, \log p\})$ (since $f > \log^3 p$ and $f \leqslant p/\log^2 p$).

The thesis of the theorem follows from Theorem 5.8 and the three cases. □

6. Discussion and future work

In this paper we presented two new algorithms, the first solves the gossip problem for synchronous, message-passing, crash-prone processors, the second solves the problem of performing a collection of tasks in a distributed setting, called *Do-All*. The gossip algorithm substantially improves the message efficiency of the best previous result. Using the new gossip algorithm as a building block, our new algorithm for the *Do-All* problem achieves better work and message complexity than any previous *Do-All* algorithms in the same model, for the full range of crashes ($f < p$).

Our techniques involve the use of conceptual communication graphs and sets of permutations with specific combinatorial properties. A future direction is to investigate how to efficiently construct permutations with the required combinatorial properties. Another direction is to extend the techniques developed in this paper to other models, for example, for synchronous restartable fail-stop processors. Note that the adversarial model with restartable processors needs to be carefully defined to eliminate the uninteresting situations where the adversary repeatedly crashes then restarts all processors, or where the crashes involving the loss of state perpetually prevent rumors from being propagated to restarted processors.

Finally, it is interesting to consider other distributed computing problems where the use of our efficient gossip algorithm can lead to better results.

References

[1] N. Alon, F. Chung, Explicit construction of linear sized tolerant networks, Discrete Math. 72 (1988) 15–19.
[2] N. Alon, H. Kaplan, M. Krivelevich, D. Malkhi, J. Stern, Scalable secure storage when half the system is faulty, in: 27th Internat. Colloq. on Automata, Languages and Programming, 2000, pp. 577–587.
[3] N. Alon, J.H. Spencer, The Probabilistic Method, second ed., Wiley, New York, 2000.
[4] R.J. Anderson, H. Woll, Algorithms for the certified Write-All problem, SIAM J. Comput. 26 (5) (1997) 1277–1283.
[5] B. Chlebus, R. De Prisco, A.A. Shvartsman, Performing tasks on restartable message-passing processors, Distributed Comput. 14 (1) (2001) 49–64.
[6] B.S. Chlebus, L. Gasieniec, D.R. Kowalski, A.A. Shvartsman, Bounding work and communication in robust cooperative computation, 16th Internat. Symp. on Distributed Computing, 2002, pp. 295–310.
[7] B.S. Chlebus, D.R. Kowalski, Gossiping to reach consensus, 14th Symp. on Parallel Algorithms and Architectures, 2002, pp. 220–229.
[8] H. Davenport, Multicative Number Theory, second ed., Springer, 1980.
[9] R. De Prisco, A. Mayer, M. Yung, Time-optimal message-efficient work performance in the presence of faults, 13th Symp. on Principles of Distributed Computing, 1994, pp. 161–172.
[10] C. Dwork, J. Halpern, O. Waarts, Performing work efficiently in the presence of faults, SIAM J. Comput. 27 (5) (1998) 1457–1491.
[11] Z. Galil, A. Mayer, M. Yung, Resolving message complexity of byzantine agreement and beyond, 36th Symp. on Foundations of Computer Science, 1995, pp. 724–733.
[12] Ch. Georgiou, D. Kowalski, A.A. Shvartsman, Efficient gossip and robust distributed computation, 17th Internat. Symp. on Distributed Computing, 2003, pp. 224–238.
[13] Ch. Georgiou, A. Russell, A.A. Shvartsman, The complexity of synchronous iterative Do-All with crashes, Distributed Comput. 17 (1) (2004) 47–63.
[14] P.C. Kanellakis, A.A. Shvartsman, Efficient parallel algorithms can be made robust, Distributed Comput. 5 (4) (1992) 201–217.
[15] A. Lubotzky, R. Phillips, P. Sarnak, Ramanujan graphs, Combinatorica 8 (1988) 261–277.
[16] G. Malewicz, A. Russell, A.A. Shvartsman, Distributed cooperation during the absence of communication, 14th Internat. Symp. Distributed Computing on 2000, 119–133.
[17] A. Pelc, Fault-tolerant broadcasting and gossiping in communication networks, Networks 28 (1996) 143–156.
[18] R.D. Schlichting, F.B. Schneider, Fail-stop processors: an approach to designing fault-tolerant computing systems, ACM Trans. Comput. Systems 1 (3) (1983) 222–238.
[19] E. Upfal, Tolerating a linear number of faults in networks of bounded degree, Inform. and Comput. 115 (1994) 312–320.