

Efficient Gossip and Robust Distributed Computation^{*}

Chryssis Georgiou¹, Dariusz R. Kowalski^{1,2}, and Alex A. Shvartsman^{1,3}

¹ Department of Computer Science and Engineering,
University of Connecticut, Storrs, CT 06269, USA

² Instytut Informatyki, Uniwersytet Warszawski, Warsaw, Poland

³ Computer Science and Artificial Intelligence Laboratory,
Massachusetts Institute of Technology, Cambridge, MA 02139, USA
{cg2,kowalski,aas}@cse.uconn.edu

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 of the new algorithm 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 $\Theta(p \cdot n)$ 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.

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

^{*} This research was supported by the NSF Grants 9988304, 0121277, and 0311368. The work of the second author was supported by the NSF-NATO Award 0209588. The work of the third author was supported by the NSF CAREER Award 9984778.

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 a synchronous message passing system and under an 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, Halpern and Waarts [7]: 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 [5]. 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 [13]. 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 [5] 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 $O(p \text{ polylog } p)$. 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* [11]), and according to their communication efficiency. Trivial solutions to *Do-All* are obtained by having each processor obviously 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, Halpern and Waarts [7], by De Prisco, Mayer and Yung [6], and by Galil, Mayer and Yung [8]. (The analysis in [7] uses task-oriented work that allows processors to idle.) The algorithm by Galil *et al.* [8]

has work $O(n + fp)$ and message complexity $O(fp^\varepsilon + p \min\{f + 1, \log p\})$. 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 [6]. The solution of Chlebus *et al.* [3,9] beats this lower bound by using multiple coordinators. It 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 $f \leq p / \log p$, 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*.

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.* [4] 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. The algorithms of Anderson and Woll [2] for the shared-memory model and of Malewicz *et al.* [12] 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 gossip algorithm substantially improves on the message complexity $M = O(p^{1.77})$ of the previously best known algorithm of Chlebus and Kowalski [5], that has the same asymptotic time complexity.

2. We demonstrate the strength of our gossip result 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.* cited earlier [8], while obtaining the same message complexity. We also improve on the result of Chlebus *et al.* [4] that has $W = O(n + p^{1.77})$ and $M = O(p^{1.77})$. Unlike the algorithm of Chlebus *et al.* [3] 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 [5,4].

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. Finally, in Section 6 we discuss future work. Complete proofs appear in the full version of the paper [10].

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 $[p] = \{1, 2, \dots, p\}$. 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. A processor may crash at any moment and once crashed it does not restart. We let an omniscient *adversary* impose failures, and we use the term *failure pattern* to denote the set of crashes. A *failure model* \mathcal{F} is the set of all failure patterns. 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. To guarantee progress, we assume that $f < p$. Formally, $|F| \leq f < p$, for any $F \in \mathcal{F}$. 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. 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 failed processor v , non-faulty processor w either knows that v has failed, or w knows v 's rumor.
- (2) *Termination:* Every non-faulty processor eventually terminates its protocol.

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

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 (tid) from the set $\mathcal{T} = [n]$. 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 processor knows this.*
- (2) *Termination: Every non-faulty processor eventually terminates its protocol.*

We let $Do-All(n, p, f)$ stand for *Do-All* for n tasks, p processors, and any pattern of crashes with $|F| \leq f < p$. We consider $Do-All(n, p, f)$ as **solved** when all tasks are done and at least one processor knows about it.

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

Definition 1. *If a p -processor algorithm solves a problem in the presence of a failure pattern F in the model \mathcal{F} by time $\tau(p, F)$, then its time complexity T is defined as $T(p, f) = \max_{F \in \mathcal{F}, |F| \leq f} \{\tau(p, F)\}$.*

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 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. *If a p -processor algorithm solves a problem in the presence of a failure pattern F in the model \mathcal{F} by time $\tau(p, F)$, then its message complexity M is defined as $M(p, f) = \max_{F \in \mathcal{F}, |F| \leq f} \{\sum_{i \leq \tau(p, F)} M_i(p, F)\}$.*

Where message complexity M depends on the size of the problem n , we similarly define it 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 surviving step i of the computation.

¹ The complexity results in this paper, except for the results in Section 5.3, also hold for a stronger definition of the termination time that requires that each non-faulty processor terminates its protocol.

Definition 3. If a p -processor algorithm solves a problem of size n in the presence of a failure pattern F in the model \mathcal{F} by time $\tau(n, p, F)$, then its work W is defined as $W(n, p, f) = \max_{F \in \mathcal{F}, |F| \leq f} \{ \sum_{i \leq \tau(n, p, F)} P_i(n, p, F) \}$.

3 Combinatorial Tools

We now develop tools used to control the message complexity of our gossip algorithm.

Communication Graphs. We now describe communication graphs—conceptual data structures that constrain communication patterns. Here processor v can send a message to any processor w that v considers to be non-faulty and that is a neighbor of v according to the communication graph. 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. For a subgraph G_Q of G induced by Q ($Q \subseteq V$), 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 f denote a positive integer with the property that even if f nodes are removed from V , the graph induced by the remaining nodes will guarantee “progress in communication”. Let G_{V_i} be the subgraph of G induced by the sets V_i of nodes (corresponding to processors that haven’t failed by step i). We assume that sets V_i have the following two properties: $V_{i+1} \subseteq V_i$ and $|V_i| \geq p - f$.

Intuitively “progress in communication” according to graph G is achieved if there is at least one “good” connected component G_{Q_i} of G_{V_i} , which evolves suitably with time and satisfies the following properties: (i) the component contains “sufficiently many” nodes so that collectively that 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) $Q_{i+1} \subseteq Q_i$ to guarantee consistency of computation. We formalize the above intuitive definition of G_{Q_i} as follows:

Definition 4. Graph $G = (V, E)$ has the Compact Chain Property $CCP(p, f, \varepsilon)$, if:

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

We prove existence of graphs satisfying CCP for some parameters.

Lemma 1. For $p > 2$, every $f < p$ and constant $\varepsilon > 0$, there is a graph G of $O(p)$ nodes satisfying $CCP(p, f, \varepsilon)$.

Sets of Permutations and their Properties. We deal with sets of permutations that satisfy certain properties. These permutations are used by the processors in the gossip algorithm to decide the subset of processors they will communicate their information with at a given point of the computation. Consider the group S_t of all permutations on set $\{1, \dots, t\}$, with the composition operation \circ , and identity e_t . For permutation $\pi = \langle \pi(1), \dots, \pi(t) \rangle$ 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., $|\{j < i : \pi(j) > \pi(i)\}| < d$.

Let \mathcal{Y} and Ψ , $\mathcal{Y} \subseteq \Psi$, be two sets containing permutations from S_t . For every σ in S_t , let $\sigma \circ \mathcal{Y}$ denote the set of permutation $\{\sigma \circ \pi : \pi \in \mathcal{Y}\}$. For given permutation π , let (d) -LRM(π) denote the number of d -left-to-right maxima in π . Now we define the notion of *surfeit*². For a given \mathcal{Y} and permutation $\sigma \in S_t$, let $(d, |\mathcal{Y}|)$ -Surf(\mathcal{Y}, σ) be equal to $\sum_{\pi \in \mathcal{Y}} (d)$ -LRM($\sigma^{-1} \circ \pi$). We then define the (d, q) -surfeit on set Ψ as (d, q) -Surf(Ψ) = $\max\{(d, |\mathcal{Y}|)$ -Surf(\mathcal{Y}, σ) : $\mathcal{Y} \subseteq \Psi, |\mathcal{Y}| = q, \sigma \in S_t\}$.

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

Theorem 1. *For a random set of p permutations Ψ from S_t , the event “for every positive integers d and $q \leq p$, (d, q) -Surf(Ψ) $> t \ln t + 10qd \ln(t + p)$ ” holds with probability at most $e^{-t \ln t \cdot \ln(9/e^2)}$.*

Using the probabilistic method [1] we obtain the following result.

Corollary 1. *There is a set of p permutations Ψ from S_t such that, for every positive integers d and $q \leq p$, (d, q) -Surf(Ψ) $\leq t \ln t + 10qd \ln(t + p)$.*

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 $\text{GOSSIP}_\varepsilon$, improves on the algorithm in [5]. The improvement are obtained by using the better properties of communication graphs described in Lemma 1, and by using many phases instead of the two phases in [5]. The challenges motivating our techniques are: (i) how to assure low communication during every phase, and (ii) how to switch between phases without a “huge complexity hit”.

4.1 Description of Algorithm $\text{GOSSIP}_\varepsilon$

Suppose constant $0 < \varepsilon < 1/3$ is given. 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 failed. 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. In the update stage processors update their local knowledge regarding other processors’ rumor (known/unknown) and condition (failed/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 failed. Epoch $\lceil 1/\varepsilon \rceil - 1$ is the terminating epoch where each processor sends a message to all the processors that it haven’t heard about, requesting their rumor.

² We will show that *surfeit* relates to the redundant activity in our algorithms i.e., “overdone” activity, or literally “surfeit”.

Iterating epochs	Terminating epoch ($\lceil 1/\varepsilon \rceil - 1$)
for $\ell = 1$ to $\lceil 1/\varepsilon \rceil - 2$ do if BUSY is empty then set <i>status</i> to idle ; $\text{NEIGHB} = \{v : v \in \text{ACTIVE} \wedge v \in N_{G_\ell}\}$; repeat $\alpha \log^2 p$ times update stage; communication stage;	update stage; if <i>status</i> = collector then send $\langle \text{ACTIVE}, \text{BUSY}, \text{RUMORS}, \text{call} \rangle$ to each processor in WAITING ; receive messages; send $\langle \text{ACTIVE}, \text{BUSY}, \text{RUMORS}, \text{reply} \rangle$ to each processor in ANSWER ; receive messages; update RUMORS ;

Fig. 1. Algorithm $\text{GOSSIP}_\varepsilon$. Code for processor v .

The pseudocode of the algorithm is given in Figure 1 (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). The correctness of the algorithm is shown in the full paper [10].

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 1. Moreover, each processor v is associated with the variable status_v . Initially $\text{status}_v = \text{collector}$ (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 $\text{status}_w = \text{informer}$ 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 contains the pids of the processors that v considers to be non-faulty. Initially, list ACTIVE_v contains all p pids. List BUSY_v contains the pids of the processors that v consider as collectors. Initially list BUSY_v contains all pids except from v , *permuted according to* π_v . List WAITING_v 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 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 failed. Initially list RUMORS_v contains the pair (v, rumor_v) .

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 , $\ell \leq \lceil 1/\varepsilon \rceil - 2$, that satisfies property $\text{CCP}(p, p - p^{1-\ell\varepsilon}, \varepsilon)$ (see Definition 4). 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* = **call**, processor v requires an answer from processor w and we refer to such message as a *call-message*. When *type* = **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 ANSWER_v contains the pids of the processors that v received a call-message. Initially set ANSWER_v is empty. Set CALLING_v contains the pids of the processors that v will send a call-message. Initially CALLING_v is empty. Set NEIGHB_v 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 ($\text{NEIGHB}_v = \{w : w \in \text{ACTIVE}_v \wedge w \in N_{G_\ell}(v)\}$). Initially, NEIGHB_v contains all neighbors of v (all nodes in $N_{G_1}(v)$).

Communication stage. In this stage the processors communicate in an attempt to obtain information from other processors. This stage contains 4 *sub-stages*. In the first sub-stage, every processor v that is either a collector or an informer (i.e., $\text{status}_v \neq \text{idle}$) sends message $\langle \text{ACTIVE}_v, \text{BUSY}_v, \text{RUMORS}_v, \text{call} \rangle$ to every processor in CALLING_v . The idle processors do not send any messages in this sub-stage. In the 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 . In the third sub-stage, every processor (regardless of its status) responds to each processor that received a call-message from. Specifically, processor v sends message $\langle \text{ACTIVE}_v, \text{BUSY}_v, \text{RUMORS}_v, \text{reply} \rangle$ to the processors in ANSWER_v and empties ANSWER_v . In the fourth and final sub-stage, the processors receive the responses to their call-messages.

Update stage. In this stage each processor v updates its local knowledge based on the messages it received in the *last communication stage*. If $\text{status}_v = \text{idle}$, then v idles. We now present the six **update rules** and their processing. Note that the rules are not disjoint, but 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 , (ii) if both (w, rumor_w) and (w, \perp) are in RUMORS_v , then v removes (w, \perp) from RUMORS_v , and (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 .

(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 $\text{BUSY}_{v'}$, 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 $\text{ACTIVE}_{v'}$, 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 $\text{sizeof}(\text{WAITING}_v) < p^{(\ell+1)\varepsilon}$) 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 $\text{sizeof}(\text{BUSY}_v) < p^{(\ell+1)\varepsilon}$) and all pids of set NEIGHB_v .

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 to every processor that is in WAITING_v (containing pids of the processors whose rumor v does not know or processors that failed). Then every processor receives the call-messages sent by the other processors. Next, every processor that received a call-message sends its local knowledge to the sender. Finally each processor v updates RUMORS_v based on any received information.

4.2 Analysis of Algorithm $\text{GOSSIP}_\varepsilon$

For simplicity we assume that $\lceil 1/\varepsilon \rceil = 1/\varepsilon$. Consider some set Q_ℓ , $|Q_\ell| \geq p^{1-\ell\varepsilon}$, of processors that are not idle at the beginning of epoch ℓ and survive epoch ℓ . Let $Q'_\ell \subseteq Q_\ell$ be such that $|Q'_\ell| \geq |Q_\ell|/7$ and the diameter of the subgraph induced by Q'_ℓ is at most $31 \log p$. Q'_ℓ exists because of Lemma 1 applied to graph G_ℓ and set Q_ℓ (chains have size 2). We let $d = (31 \log p + 1)p^{(\ell+1)\varepsilon}$. For any processor v , let $\text{CALL}_v = \text{CALLING}_v \setminus \text{NEIGHB}_v$. We will be referring to the call-messages sent to the processors whose pids are in CALL as *progress-messages*.

Lemma 2. *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 $(d, |Q'_\ell|)\text{-Surf}(\Psi)$.*

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

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 2 and Corollary 1 we show the following:

Lemma 3. *In any execution of algorithm $\text{GOSSIP}_\varepsilon$, the invariant I_ℓ holds for any epoch $\ell = 1, \dots, 1/\varepsilon - 2$.*

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

Proof. First we show the bound on time. Observe that each update and communication stage takes $O(1)$ time. Therefore each of the first $1/\varepsilon - 2$ epochs takes $O(\log^2 p)$ 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 worse case $O(\log^2 p)$. We now show the bound on messages. From Lemma 3 we have that for every $1 \leq \ell < 1/\varepsilon - 2$, 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 $4(\alpha \log^2 p) \cdot (p^{1-\ell\varepsilon} p^{(\ell+2)\varepsilon}) \leq 4\alpha p^{1+2\varepsilon} \log^2 p \leq 4\alpha p^{1+3\varepsilon}$. After epoch $1/\varepsilon - 2$ there are, per $I_{1/\varepsilon-2}$, at most $p^{2\varepsilon}$ processors having list `WAITING` not empty. In epoch $1/\varepsilon - 1$ 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 $O(p^{1+3\varepsilon})$.

5 The Do-All Algorithm

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

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 $\text{GOSSIP}_\varepsilon$ 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 for a phase of epoch ℓ of the algorithm is given in Figure 2 (again we assume that every **if-then** has an implicit **else** containing no-ops as needed). The correctness of the algorithm is shown in the full paper [10].

<p>Work stage</p> <pre> repeat T_ℓ times if TASK not empty then perform task whose id is first in TASK; remove task's id from TASK; elseif TASK empty and done = false then set done to true; if TASK empty and done = false then set done to true; </pre>	<p>Gossip stage</p> <pre> run $\text{GOSSIP}_{\varepsilon/3}$ with rumor = (TEMP, PROC, done); if done = true and done_w = true for all w received rumor from then TERMINATE; else update TASK and PROC; </pre>
---	---

Fig. 2. A phase of epoch ℓ of algorithm DOALL_ε . Code for processor v .

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 $\text{TASK}_v = \langle 1, \dots, n \rangle$ and $\text{PROC}_v = \langle 1, \dots, p \rangle$. 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 $[n]$.³ 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 $T_\ell = \left\lceil \frac{n+p \log^3 p}{\frac{p}{2^t} \log p} \right\rceil$ 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`.

Gossip stage. Here processors execute algorithm $\text{GOSSIP}_{\varepsilon/3}$ using their local knowledge as the rumor, i.e., for processor v , $\text{rumor}_v = (\text{TASK}_v, \text{PROC}_v, \text{done}_v)$. 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 failed (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 compare 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 *does not* change to `true`. It will change in the next work stage (we do this for technical reasons).

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 turned 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 $\text{done}_v = \text{true}$, 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).

³ This is distinct from the set of permutation on $[p]$ required by the gossip algorithm.

Remark 1. In the complexity analysis of the algorithm we first assume that $n \leq p^2$ 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 start of algorithm DOALL_ε , 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 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. Consider phase i in epoch ℓ . For a given failure pattern F , let $V_i(F)$ denote the set of processors that 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 $p_{i+1}(F) \geq p_i(F)/2$, we say that i is a *reliable* minority phase. If phase i is a reliable minority phase and $u_{i+1}(F) \leq u_i(F) - \frac{1}{4}p_{i+1}(F)T_\ell$, 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 $u_{i+1}(F) \leq \frac{3}{4}u_i(F)$, 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 Figure 3.

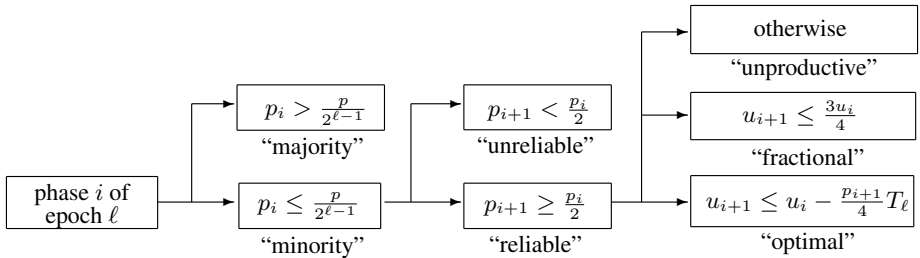


Fig. 3. Classification of a phase i of epoch ℓ ; the failure pattern F is implied.

Our goal is to choose a set Ψ of permutations 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 4 and 5), and finally use the probabilistic method to obtain an existential deterministic solution.

Lemma 4. *Let Q be a fixed nonempty subset of processors. 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 $u_i(F) - u_{i+1}(F) \geq \min\{u_i(F), |Q|T_\ell\}/4$,” is at least $1 - 1/e^{\Omega(|Q|T_\ell)}$.*

Lemma 5. *Assume $n \leq p^2$. There exists a constant integer $\beta > 0$ such that for every phase i of epoch ℓ , for any epoch ℓ , 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 $e^{-\Omega(p \log^3 p)}$, and (b) the probability that phase i is a minority reliable unproductive phase under some failure pattern F is at most $e^{-\Omega(T_\ell)}$.*

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

Proof. (Sketch.) We first consider the case $n \leq p^2$ and then we show the result for the case $n > p^2$. The idea of the proof is as follows: using Lemmas 4 and 5 we reason about the probability of a phase i of an epoch ℓ belonging to one of the classes illustrated in Figure 3, and about the work that phase i contributes to the total work, depending on its classification. We then reason that a set of permutations Ψ and a constant integer $\beta > 0$ exist so that the complexity bounds are as desired. See the full paper [10] for details.

5.3 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 $\text{DOALL}_{\varepsilon/2}$. We add two new epochs, called epoch -1 and epoch 0 . We call the new algorithm, algorithm $\text{DOALL}'_\varepsilon$.

Epoch -1 of algorithm $\text{DOALL}'_\varepsilon$ is based on the check-pointing algorithm from [6], where the check-pointing and the synchronization procedures are taken from [8]. We refer to this algorithm as 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 number of failures is small, mainly concerning the case $f \leq \log^3 p$. Hence we execute DGMY only until step $a \cdot (n/p + \log^3 p)$, for some constant a such that early-stopping condition of DGMY holds for every $f \leq \log^3 p$. We call this execution $\text{DGMY}(a \cdot (n/p + \log^3 p))$.

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

Algorithm $\text{GOSSIP}'_{\varepsilon/3}$ is obtained by adding a new epoch in $\text{GOSSIP}_{\varepsilon/3}$ and by slightly modifying the remaining epochs of $\text{GOSSIP}_{\varepsilon/3}$. We show that $\text{GOSSIP}'_{\varepsilon/3}$ solves the $\text{Gossip}(p, f)$ problem with time complexity $T = O(\log^2 p)$ and message complexity $M = O(p)$ when $f \leq \frac{p}{\log^2 p}$, and with $T = O(\log^2 p)$ and $M = O(p^{1+3\varepsilon/4})$ otherwise. Details can be found in the full version of the paper [10].

Finally we show the following result for algorithm $\text{DOALL}'_\varepsilon$, using Theorem 3, the analysis of epochs -1 and 0 , and the complexity of algorithm $\text{GOSSIP}'_{\varepsilon/3}$.

Theorem 4. *There exists a set of permutations Ψ and a constant integer $\beta > 0$ such that algorithm $\text{DOALL}'_\varepsilon$ solves the $\text{Do-All}(n, p, f)$ problem with work $W = O(n + p \cdot \min\{f + 1, \log^3 p\})$ and message complexity $M = O(fp^\varepsilon + p \min\{f + 1, \log p\})$.*

6 Discussion and Future Work

In this paper we presented two contributions. We improved the previously best known algorithm that solves the gossip problem for synchronous, message-passing, crash-prone processors. Using our new gossip algorithm we developed a new algorithm for the *Do-All* problem. Our algorithm 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. It is also interesting to consider other distributed computing problems where the use of our efficient gossip algorithm can lead to better results.

References

1. N. Alon and J.H. Spencer. *The Probabilistic Method*. J. Wiley and Sons, Inc., second edition (2000)
2. R.J. Anderson and H. Woll. Algorithms for the certified Write-All problem. *SIAM Journal of Computing*, Vol. 26 5 (1997) 1277–1283
3. B. Chlebus, R. De Prisco and A.A. Shvartsman. Performing tasks on restartable message-passing processors. *Distributed Computing*, Vol. 14 1 (2001) 49–64
4. B.S. Chlebus, L. Gasieniec, D.R. Kowalski and A.A. Shvartsman. Bounding work and communication in robust cooperative computation. *16th International Symposium on Distributed Computing* (2002) 295–310
5. B.S. Chlebus and D.R. Kowalski. Gossiping to reach consensus. *14th Symposium on Parallel Algorithms and Architectures* (2002) 220–229
6. R. De Prisco, A. Mayer and M. Yung. Time-optimal message-efficient work performance in the presence of faults. *13th Symposium on Principles of Distributed Computing* (1994) 161–172
7. C. Dwork, J. Halpern and O. Waarts. Performing work efficiently in the presence of faults. *SIAM Journal on Computing*, Vol. 27 5 (1998) 1457–1491
8. Z. Galil, A. Mayer and M. Yung. Resolving message complexity of byzantine agreement and beyond. *36th Symp. on Foundations of Comp. Sc.* (1995) 724–733
9. Ch. Georgiou, A. Russell and A.A. Shvartsman. The complexity of synchronous iterative Do-All with crashes. *15th Int-l Symposium on Distributed Computing* (2001) 151–165
10. Ch. Georgiou, D. Kowalski and A.A. Shvartsman. Efficient gossip and robust distributed computation. <http://www.engr.uconn.edu/~aas/GKS-2003.ps>
11. P.C. Kanellakis and A.A. Shvartsman. Efficient parallel algorithms can be made robust. *Distributed Computing*, Vol. 5 4 (1992) 201–217
12. G. Malewicz, A. Russell and A.A. Shvartsman. Distributed cooperation during the absence of communication. *14th Int-l Symp. on Distr. Computing* (2000) 119–133
13. A. Pelc. Fault-tolerant broadcasting and gossiping in communication networks. *Networks*, Vol. 28 (1996) 143–156