4

Synchronous Do-All with Crashes and Point-to-Point Messaging

We now study the *Do-All* problem assuming that only point-to-point messaging is available for processors to communicate. This in contrast with the assumptions in the previous chapter, we considered the *Do-All* problem assuming that processors where assisted by an oracle or that reliable multicast was available. As one would expect, in the point-to-point messaging setting the problem becomes more challenging and different techniques need to be employed in order to obtain efficient (deterministic) algorithms for *Do-All*.

We consider *Do-All*$_{\mathcal{A}_C}(n, p, f)$, that is the *Do-All* problem for n tasks, p processors, up to f crashes, as determined by the adversary $\mathcal{A}_C$. The key in developing efficient deterministic algorithms for *Do-All* in this setting lies in the ability to share knowledge among processors efficiently. Algorithms that rely on unique coordinators or checkpointing mechanisms incur a work overhead penalty of $\Omega(n+fp)$ for f crashes; this overhead is particularly large, for large f, for example, when $f = \omega(\log^{\Theta(1)} p)$. Algorithm AN (from the previous chapter) beats this lower bound by using multiple coordinators, however it uses reliable multicast, which can be viewed as a strong assumption in some distributed settings. Therefore, we are interested in developing algorithms that do not use checkpointing or reliable multicast and that are efficient, especially for large f.

In this chapter we present a synchronous, message-passing, deterministic algorithm for *Do-All*$_{\mathcal{A}_C}(n, p, f)$. This algorithm has total-work complexity $O(n + p \log^3 p)$ and message complexity $M(p^{1+2\varepsilon})$, for any $\varepsilon > 0$. Thus, the work complexity of this algorithm beats the above mentioned lower bound (for $f = \omega(\log^3 p)$) and it is comparable to that of algorithm AN–however it uses simple point-to-point messaging. The algorithm does not use coordinator or checkpointing strategies to implement information sharing among processors. Instead, it uses an approach where processors share information using an algorithm developed to solve the *gossip problem* in synchronous message-passing systems with processor crashes. To achieve messaging efficiency, 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. Processors send messages based on permutations with certain properties.

Chapter structure.

In Section 4.1 we define the gossip problem and relevant measures of efficiency. In Section 4.2 we present combinatorial tools that are used in the analysis of the gossip and *Do-All* algorithms. In Section 4.3 we present a gossip algorithm, show its correctness, and perform its complexity analysis. In Section 4.4 we present the *Do-All* algorithm itself, show its correctness, and give complexity analysis. We discuss open problems in Section 4.5.

4.1 The Gossip Problem

The *Gossip* problem is considered one of the fundamental problems in distributed computing and it is normally stated as follows: each processor has a distinct piece of information, called a *rumor*, and the goal is for each processor to learn all rumors. In our setting, where we consider processor crashes, it might not always be possible to learn the rumor of a processor that crashed, since all the processors that have learned the rumor of that processor might have also crashed in the course of the computation. Hence, we consider a variation of the traditional gossip problem. We require that every non-faulty processor learns the following about each processor v: either the rumor of v or that v has crashed. It is important to note that we do not require for the non-faulty processors to reach agreement: if a processor crashes then some of the non-faulty processors may get to learn its rumor while others may only learn that it has crashed.

Formally, we define the *Gossip* problem with crash-prone processors, as follows:

Definition 4.1. *The Gossip problem: 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 processor 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 terminates its protocol.

We let $\mathit{Gossip}_{\mathcal{A}_C}(p, f)$ stand for the *Gossip* problem for p processors (and p rumors) and adversary $\mathcal{A}_C$ constrained to adversarial patterns of weight less or equal to f.

We now define the measures of efficiency we use in studying the complexity of the *Gossip* problem. We measure the efficiency of a *Gossip* algorithm

in terms of its *time complexity* and *message complexity*. Time complexity is measured as the number of parallel steps taken by the processors until the *Gossip* problem is *solved*. The *Gossip* problem is said to be solved at step τ, if τ is the first step where the correctness condition is satisfied and at least one (non-faulty) processor terminates its protocol. More formally:

Definition 4.2 (time complexity). *Let* A *be an algorithm that solves a problem with p processors under adversary $\mathcal{A}$. If execution $\xi \in \mathcal{E}(\mathrm{A}, \mathcal{A})$, where $\|\xi|_{\mathcal{A}}\| \leq f$, solves the problem by time $\tau(\xi)$, then the* time complexity *T of algorithm* A *is:*

$$T = T_{\mathcal{A}}(p, f) = \max_{\xi \in \mathcal{E}(\mathrm{A}, \mathcal{A}),\ \|\xi|_{\mathcal{A}}\| \leq f} \{\tau(\xi)\}.$$

The message complexity is defined as in Definition 2.6 where the size of the problem is p: it is measured as the total number of point-to-point messages sent by the processors until the problem is solved. As before, when a processor communicates using a multicast, its cost is the total number of point-to-point messages.

4.2 Combinatorial Tools

We present tools used to control the message complexity of the gossip algorithm presented in the next section.

4.2.1 Communication Graphs

We first describe *communication graphs* — conceptual data structures that constrain communication patterns.

Informally speaking, the computation begins with a communication graph that contains all nodes, where each node represents a processor. 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). 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 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 haven't crashed by step i of a given execution. Hence $V_{i+1} \subseteq V_i$ (since processor do not restart). Also, each $|V_i| \geq p - f$, since no more than $f < p$ processors may crash in a given execution. Let G_{Q_i} denote a component of G_{V_i} where $Q_i \subseteq V_i$.

To formulate the the notion of a "good" component G_{Q_i} we define a property, called *Compact Chain Property* (CCP):

Definition 4.3. *Graph* $G = (V, E)$ *has the* Compact Chain Property *CCP*(p, f, ε)*, if:*
I. *The maximum degree of* G *is at most* $\left(\frac{p}{p-f}\right)^{1+\varepsilon}$,
II. *For a given sequence* $V_1 \supseteq \ldots \supseteq V_k$ ($V = V_1$)*, where* $|V_k| \geq p - f$*, there is a sequence* $Q_1 \supseteq \ldots \supseteq Q_k$ *such that for every* $i = 1, \ldots, k$*:*
(a) $Q_i \subseteq V_i$,
(b) $|Q_i| \geq |V_i|/7$*, and*
(c) *the diameter of* G_{Q_i} *is at most* $31 \log p$.

The following shows existence of graphs satisfying *CCP* for some parameters.

Lemma 4.4. *For* $p>2$*, every* $f<p$*, and constant* $\varepsilon>0$*, there is a graph* G *of* $O(p)$ *nodes satisfying property CCP*(p, f, ε).

4.2.2 Sets of Permutations

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 symmetric group $\mathcal{S}_t$ 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 $\mathcal{S}_t$, 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., $|\{\pi(j) : \pi(j) > \pi(i) \wedge j < i\}| < d$. For a given permutation π, let (d)-LRM(π) denote the number of d-left-to-right maxima in π.

Let Υ and Ψ, $\Upsilon \subseteq \Psi$, be two sets containing permutations from $\mathcal{S}_t$. For every σ in $\mathcal{S}_t$, let $\sigma \circ \Upsilon$ denote the set of permutations $\{\sigma \circ \pi : \pi \in \Upsilon\}$. Now we define the notion of *surfeit*. (We will show that *surfeit* relates to the redundant activity in our algorithms, i.e., "overdone" activity, or literally

"surfeit".) For a given Υ and permutation $\sigma \in \mathcal{S}_t$, 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$(\Psi) = \max\{(d, q)\text{-Surf}(\Upsilon, \sigma) : \Upsilon \subseteq \Psi \wedge |\Upsilon| = q \wedge \sigma \in \mathcal{S}_t\}$.

The following results are known for (d, q)-surfeit.

Lemma 4.5. *Let Υ be a set of q random permutations on set $\{1, \ldots, t\}$. For every fixed positive integer d, the probability that (d, q)-Surf$(\Upsilon, \mathbf{e}_t) > t \ln t + 10qd \ln(t + p)$ is at most $e^{-[t \ln t + 9qdH_{t+p}] \ln(9/e)}$.*

Theorem 4.6. *For a random set of p permutations Ψ from $\mathcal{S}_t$, the event "for every positive integers d and $q \leq p$, (d, q)-Surf$(\Psi) > t \ln t + 10qd \ln(t + p)$" holds with probability at most $e^{-t \ln t \cdot \ln(9/e^2)}$.*

Using the probabilistic method we obtain the following result.

Corollary 4.7. *There is a set of p permutations Ψ from $\mathcal{S}_t$ such that, for every positive integers d and $q \leq p$, (d, q)-Surf$(\Psi) \leq t \ln t + 10qd \ln(t + p)$.*

The efficiency of the gossip algorithm (and hence the efficiency of a *Do-All* algorithm that uses such gossip) relies on the existence of the permutations in the thesis of the corollary (however the algorithm is correct for any permutations). These permutations can be efficiently constructed.

4.3 The Gossip Algorithm

We now present the gossip algorithm, called GOSSIP$_\varepsilon$.

4.3.1 Description of Algorithm 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.

The pseudocode of the algorithm is given in Figure 4.1 (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; this is used to ensure the synchrony of the system). The details of the algorithm are explained in the rest of this section.

Initialization

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

Iterating epochs

for $\ell = 1$ **to** $\lceil 1/\varepsilon \rceil - 2$ **do**
  if BUSY_v is empty **then** set $status_v$ to idle;
  $\text{NEIGHB}_v = \{w : w \in \text{ACTIVE}_v \wedge w \in N_{G_\ell}(v) \setminus \{v\}\}$;
  repeat $\alpha \log^2 p$ times *% iterating phases*
    update stage;
    communication stage;

Terminating epoch $(\lceil 1/\varepsilon \rceil - 1)$

update stage;
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;

Fig. 4.1. Algorithm GOSSIP$_\varepsilon$, stated for processor v; $\pi_v(i)$ denotes the i^{th} element of permutation π_v.

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 4.7. Moreover, each processor v is associated with the variable $status_v$. Initially $status_v =$ `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 $status_w =$ `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 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 $CCP(p, p - p^{1-\ell\varepsilon}, \varepsilon)$ (see Definition 4.3 and Lemma 4.4). 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 $type = \texttt{call}$, processor v requires an answer from processor w and we refer to such message as a *call-message*. When $type = \texttt{reply}$, 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: it contains the pids of the processors that v received a call-message. Initially set ANSWER_v is empty.
- Set CALLING_v: it contains the pids of the processors that v will send a call-message. Initially CALLING_v is empty.
- 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 ($\text{NEIGHB}_v = \{w : w \in \text{ACTIVE}_v \wedge w \in N_{G_\ell}(v)\}$). 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 *four sub-stages*:

- First sub-stage: every processor v that is either a collector or an informer (i.e., $status_v \neq \texttt{idle}$) sends message $\langle \text{ACTIVE}_v, \text{BUSY}_v, \text{RUMORS}_v, \texttt{call} \rangle$ 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 $\langle \text{ACTIVE}_v, \text{BUSY}_v, \text{RUMORS}_v, \texttt{reply} \rangle$ to the processors in ANSWER_v and empties ANSWER_v.
- Fourth 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*[1]. If $status_v = \texttt{idle}$, 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 $sizeof(\text{WAITING}_v) < p^{(\ell+1)\varepsilon}$) and all pids of set NEIGHB_v, and

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

(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$_v$.

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$_v$ (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$_v$.

4.3.2 Correctness of Algorithm GOSSIP$_\varepsilon$

We show that algorithm GOSSIP$_\varepsilon$ solves the *Gossip*$_{\mathcal{A}_C}(p, f)$ 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 4.8. *In any execution of algorithm* 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, w is not in NEIGHB$_v$ and v received a message from a processor v' so that w is not in ACTIVE$_{v'}$,
(b) during the communication stage of phase s, 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 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 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 $status_w \neq \texttt{idle}$, w sends a message to its neighbors, including v. If $status_w = \texttt{idle}$, 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.
Neither Case (a) is 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 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 4.9. *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* $status_w = \texttt{collector}$*, 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 $status_w = \texttt{collector}$. 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' so that 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 4.10. *In any execution of algorithm* $\text{GOSSIP}_\varepsilon$ *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 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 $\text{RUMORS}_{v'}$ (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 $\text{GOSSIP}_\varepsilon$.

Theorem 4.11. *By the end of epoch* $\lceil 1/\varepsilon \rceil - 1$ *of any execution of algorithm* $\text{GOSSIP}_\varepsilon$*, 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 4.8, 4.9, and 4.10 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 this last update stage is no different from the update stage 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 4.10 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 4.7. The algorithm is correct for any set of permutations of $[p]$.

4.3.3 Analysis of Algorithm GOSSIP$_\varepsilon$

Consider some set V_ℓ, $|V_\ell| \geq p^{1-\ell\varepsilon}$, of processors that are not idle at the beginning of epoch ℓ and do not fail by the end of epoch ℓ. Let $Q_\ell \subseteq V_\ell$ be such that $|Q_\ell| \geq |V_\ell|/7$ and the diameter of the subgraph induced by Q_ℓ is at most $31 \log p$. Q_ℓ exists because of Lemma 4.4 applied to graph G_ℓ and set V_ℓ.

For any processor v, let $\text{CALL}_v = \text{CALLING}_v \setminus \text{NEIGHB}_v$. 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 $d \geq (31 \log p + 1) \cdot |\text{CALL}|$.

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

Lemma 4.12. *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 $i < j \leq p$. Let τ_i (τ_j) be the time step in epoch ℓ where some processor in Q_ℓ hears about $\sigma(i)$ ($\sigma(j)$) the first time among the processors in Q_ℓ. Then $\tau_i \leq \tau_j$." (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$(\Upsilon, \sigma) \leq (d, |Q_\ell|)$-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 none of processor $v' \in Q_\ell$ had sent a progress-message to w before $31 \log p$ phases, and consequently position of processor w in permutation π_v is at most $d - |\text{CALL}| \leq d - p^{(\ell+1)\varepsilon}$ greater than 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 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 $j \leq i$ 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 $j \leq i$ such that $(\sigma^{-1} \circ \pi_v)(j)$ is a d-lrm and j is not assigned by h before step of construction for $(v,i) \in P_v$. Let $j_1 < \ldots < j_d < i$ 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 $\frac{d}{|\text{CALL}|} = 31 \log p + 1$ 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$(\Upsilon, \sigma) \leq (d, |Q_\ell|)$-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 4.12 and Corollary 4.7 we show the following:

Lemma 4.13. *In any execution of algorithm* GOSSIP$_\varepsilon$*, the invariant* 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 4.4 and Corollary 4.7. 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 $|Q_\ell| \geq |V_\ell|/7 > p^{1-\ell\varepsilon}/7$ and the diameter of the subgraph induced by Q_ℓ is at most $31 \log p$. Applying Lemma 4.12 and Corollary 4.7 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

$$2 \cdot (31(\log p + 1)p^{(\ell+1)\varepsilon}, |Q_\ell|)\text{-Surf}(\Psi) + 31|Q_\ell|p^{(\ell+1)\varepsilon} \log p$$

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

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

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

Recall that $\alpha\log^2 p \geq 341\log^2 p$. 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 GOSSIP$_\varepsilon$.

Theorem 4.14. *Algorithm* GOSSIP$_\varepsilon$ *solves the Gossip*$_{\mathcal{A}_C}(p, f)$ *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 $\lceil 1/\varepsilon\rceil - 2$ 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 4.13 we have that for every $1 \leq \ell < \lceil 1/\varepsilon\rceil - 2$, 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 $4(\alpha\log^2 p)\cdot(p^{1-\ell\varepsilon}p^{(\ell+2)\varepsilon}) \leq 4\alpha p^{1+2\varepsilon}\log^2 p \leq 4\alpha p^{1+3\varepsilon}$. 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 $O(p^{1+3\varepsilon})$. □

4.4 The Do-All Algorithm

We now put the gossip algorithm to use by constructing a robust *Do-All* algorithm, called algorithm DOALL$_\varepsilon$.

4.4.1 Description of Algorithm DOALL$_\varepsilon$

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 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 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 4.2 (again we assume that every *if-then* has an implicit *else* containing no-ops as needed to ensure the synchrony of the system). The details are explained in the rest of this section.

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 TASK$_v = \langle 1, \ldots, n\rangle$ and 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 $[n]$. (This is distinct from the set of permutation on $[p]$ required by the gossip algorithm.) 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. These permutations can be constructed efficiently.

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 $T_\ell = \left\lceil \frac{n+p\log^3 p}{\frac{p}{2^\ell}\log p} \right\rceil$ 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

Initialization

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

Epoch ℓ

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

repeat $T_\ell = \lceil \frac{n + p \log^3 p}{\frac{p}{2^\ell} \log p} \rceil$ times *% work stage 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 $rumor_v = (\text{TASK}_v, \text{PROC}_v, done_v)$; *% gossip stage 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;

Fig. 4.2. Algorithm DOALL_ε, stated for processor v; $\pi_v(i)$ denotes the i^{th} element of permutation π_v.

sub-stage prior to sub-state T_ℓ, 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, $rumor_v = (\text{TASK}_v, \text{PROC}_v, done_v)$. 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 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 $done_v$ remains `false`. It will be set to `true` in the next work stage. This is needed for the correctness of the algorithm (see Lemma 4.19).

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 termi-

nate 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 $done_v = \texttt{true}$, 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 4.15. 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.

4.4.2 Correctness of Algorithm DOALL_ε

We show that the algorithm DOALL_ε solves the *Do-All*$_{\mathcal{A}_C}(n, p, f)$ problem correctly, meaning that the algorithm terminates with all tasks performed and all non-faulty processors are aware of this. Note that this is a stronger correctness condition than the one required by 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 4.16. *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 at 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 4.17. *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, or (ii) during the gossip stage v received $rumor_w$ from processor w in which z was not in TASK_w. The latter suggests that processor w performed task z during the work stage. Continuing in this manner it follows 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 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. The latter suggests 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. 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 4.18. *In any execution of algorithm* DOALL_ε*, if processor* v *enters a work stage of a phase* s *of epoch* ℓ *with* $done_w$ = `false`*, 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 $done_w =$ `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. Hence, 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 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 4.19. *In any execution of algorithm* DOALL$_\varepsilon$*, 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 GOSSIP$_{\varepsilon/3}$ is empty. Since v does not fail during phase s, the correctness of algorithm 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 $done_w =$ `true` and all rumors they receive contain the variable *done* set to `true`.

The above, in conjunction with the termination guarantees of algorithm 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$_\varepsilon$.

Theorem 4.20. *In any execution of algorithm* DOALL$_\varepsilon$*, the algorithm terminates with all tasks performed and all non-faulty processors being aware of this.*

Proof. By Lemma 4.16, no non-faulty processor leaves the computation, and by our model at least one processor does not crash ($f < p$). Also from Lemma 4.17 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 4.18 we have that local progress is guaranteed. The above in conjunction with the correctness of algorithm GOSSIP$_{\varepsilon/3}$ lead 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 4.19 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$_\varepsilon$ 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 $\beta > 0$.

4.4.3 Analysis of Algorithm DOALL$_\varepsilon$

We now derive the work and message complexities for algorithm DOALL$_\varepsilon$. The analysis is based on the following terminology. For the purpose of the analysis, we number globally all phases by positive integers starting from 1. Consider a phase i in epoch ℓ of an execution $\xi \in \mathcal{E}(\text{DOALL}_\varepsilon, \mathcal{A}_C)$. Let $V_i(\xi)$ denote the set of processors that are non-faulty at the beginning of phase i. Let $p_i(\xi) = |V_i(\xi)|$. Let $U_i(\xi)$ denote the set of tasks z such that z is in some list TASK$_v$, for some $v \in V_i(\xi)$, at the beginning of phase i. Let $u_i(\xi) = |U_i(\xi)|$.

Now we classify the possibilities for phase i as follows. If at the beginning of phase i, $p_i(\xi) > p/2^{\ell-1}$, 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 $p_i(\xi)/2$, i.e., $p_{i+1}(\xi) < p_i(\xi)/2$, we say that i is an *unreliable* minority phase. If $p_{i+1}(\xi) \geq p_i(\xi)/2$, we say that i is a *reliable* minority phase. If phase i is a reliable minority phase and $u_{i+1}(\xi) \leq u_i(\xi) - \frac{1}{4} p_{i+1}(\xi) T_\ell$, 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 $u_{i+1}(\xi) \leq \frac{3}{4} u_i(\xi)$, 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 4.3.

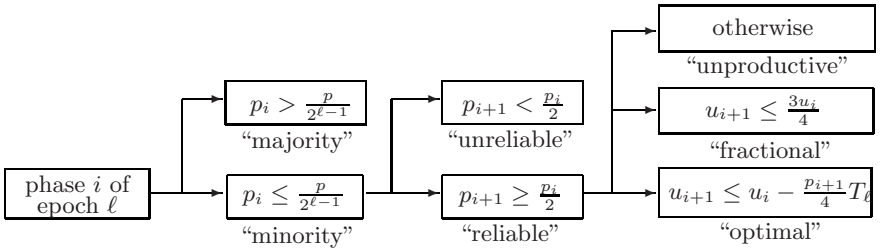

Fig. 4.3. Classification of a phase i of epoch ℓ; execution ξ is implied.

Our goal is to choose a set Ψ of permutations and a constant $\beta > 0$ such that for any execution there will be no unproductive and no majority phases. To do this we analyze sets of random permutations, prove certain properties of algorithm DOALL$_\varepsilon$ for such sets (in Lemmas 4.21 and 4.22), 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 $\min\{u_i(\xi), |Q|T_\ell\}/4$ yet unperformed tasks are performed by the processors if the phase is a minority reliable unproductive phase, all with high probability. Hence one can derandomize the choice of suitable set of permutations such that for any execution there are neither majority nor minority reliable unproductive phases.

Lemma 4.21. *Let Q be a fixed nonempty subset of processors in phase i of epoch ℓ of algorithm* DOALL$_\varepsilon$. *Then the probability of event "for every execution ξ of algorithm* DOALL$_\varepsilon$ *such that $V_{i+1}(\xi) \supseteq Q$ and $u_i(\xi) > 0$, the following inequality holds $u_i(\xi) - u_{i+1}(\xi) \geq \min\{u_i(\xi), |Q|T_\ell\}/4$," is at least $1 - 1/e^{-|Q|T_\ell/8}$.*

Proof. Let ξ be an execution of algorithm DOALL$_\varepsilon$ such that $V_{i+1}(\xi) \supseteq Q$ and $u_i(\xi) > 0$. Let $c = \min\{u_i(\xi), |Q|T_\ell\}/4$. Let $S_i(\xi)$ 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 $s_i(\xi) = |S_i(\xi)|$. Note that $S_i(\xi) \subseteq U_i(\xi)$, and that $S_i(\xi)$ describes some properties of set Q, while $U_i(\xi)$ describes some properties of set $V_i(\xi) \supseteq Q$. Consider the following cases:

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

Case 2: $s_i(\xi) > u_i(\xi) - c$. 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 the work stage of phase i. This process takes $|Q|T_\ell$ steps to be completed. Let $U_i^{(k)}(\xi)$ 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 $u_i^{(k)}(\xi) = |U_i^{(k)}(\xi)|$. Define the random variables X_k, for $1 \leq k \leq |Q|T_\ell$, as follows:

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

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

$$s_i(\xi) - \big(u_i(\xi) - u_i^{(k)}(\xi)\big) > s_i(\xi) - c \geq u_i(\xi)/2 \geq \mathit{sizeof}(\textsc{task}_v)/2,$$

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

We need to estimate the probability $\Pr[\sum X_k \geq c]$, 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 $\Pr[Y_k = 1] = 1/2$. Then the sequence $\langle X_k \rangle$ statistically dominates the sequence $\langle Y_k \rangle$, in the sense that $\Pr\big[\sum X_k \geq d\big] \geq \Pr\big[\sum Y_k \geq d\big]$, for any $d > 0$. Note that $\mathbb{E}[\sum Y_k] = |Q|T_\ell/2$ and $c \leq \mathbb{E}[\sum Y_k]/2$, hence we can apply Chernoff bound to obtain

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

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

Lemma 4.22. *Assume $n \leq p^2$ and $p \geq 2^8$. There exists a constant integer $\beta > 0$ such that for every phase i of some epoch $\ell > 1$ of any execution ξ of algorithm $\textsc{Doall}_\varepsilon$, if there is a task unperformed by the beginning of phase i then:*

(a) the probability that phase i is a majority phase is at most $e^{-p \log p}$, and
(b) the probability that phase i is a minority reliable unproductive phase 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 executions ξ such that phase i is a majority phase in ξ, according to the following equivalence relation: executions ξ_1 and ξ_2 are in the same class iff $V_{i+1}(\xi_1) = V_{i+1}(\xi_2)$. Every such equivalence class is represented by some set of processors Q of size greater than $\frac{p}{2^{\ell-1}}$, such that for every execution ξ in this class we have $V_{i+1}(\xi) = Q$. In the following claim we define conditions for β for satisfying clause (a).

Claim. *For constant $\beta = 9$ and any execution ξ in the class represented by Q, where $|Q| > \frac{p}{2^{\ell-1}}$, 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. Consider an execution ξ from a class represented by Q. Consider all steps taken by processors in Q during phase j of epoch $\ell - 1$. By Lemma 4.21, since $V_{j+1}(\xi) \supseteq Q$, we have that the probability of event "if $u_j(\xi) > 0$ then $u_j(\xi) - u_{j+1}(\xi) \geq \min\{u_j(\xi), |Q|T_{\ell-1}\}/4$," 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), and this happens 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

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

or, since $n \leq p^2$, is at most

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

Therefore there are a total of $9\log p$ productive phases, which is sufficient to perform all 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

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

since $p \geq 8$. Consequently all processors terminate by the end of phase $\beta \log p + 1$ with probability $1 - e^{-p\log p - p}$. This follows by the correctness of the gossip algorithm and the argument of Lemma 4.19, 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 of possible sets Q of processors, hence by the Claim the probability that phase i is a majority phase is at most

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

which proves clause (a) for phase i.

Now we prove clause (b) for phase i. Consider executions such that phase i in epoch ℓ is a minority reliable phase. Similarly as above, we partitions executions according to the following equivalence relation: executions ξ_1 and ξ_2 are in the same class if there is set Q such that $H = V_{i+1}(\xi_1) = V_{i+1}(\xi_2)$. Set Q is a representative of a class. By Lemma 4.21 applied to phase i and set Q we obtain that the probability that phase i is unproductive for every execution ξ such that $V_{i+1}(\xi) = Q$ is $e^{-|Q|T_\ell/8}$. Hence the probability that for any execution ξ phase i is a minority reliable unproductive phase is at most

$$\begin{aligned}\sum_{x=1}^{p/2^{\ell-1}} \binom{p}{x} \cdot e^{-xT_\ell/8} &\leq \sum_{x=1}^{p/2^{\ell-1}} 2^{x\log p} \cdot e^{-xT_\ell/8} \leq \sum_{x=1}^{p/2^{\ell-1}} e^{x\log p - xT_\ell/8} \\ &\leq e^{\log p - T_\ell/8} \cdot \frac{1}{1 - e^{\log p - T_\ell/8}} \leq e^{-T_\ell/16},\end{aligned}$$

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

Recall that epoch ℓ consists of $\beta \log p + 1$ phases for some $\beta > 0$ and that $T_\ell = \lceil \frac{n+p\log^3 p}{(p/2^\ell)\log p} \rceil$. Also by the correctness proof of algorithm DoAll$_\varepsilon$ (Theorem 4.20), the algorithm terminates in at most $O(\log p)$ epochs, hence, the algorithm terminates in at most $O(\log^2 p)$ 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 4.23. *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$_{\mathcal{A}_C}(n,p,f)$ problem with total work $S = O(n + p\log^3 p)$ and message complexity $M = O(p^{1+2\varepsilon})$.*

Proof. We show that for any execution $\xi \in \mathcal{E}(\text{DoAll}_\varepsilon, \mathcal{A}_C)$ that solves the *Do-All*$_{\mathcal{A}_C}(n,p,f)$ problem 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 4.22. We consider two cases:

Case 1: $n \leq p^2$. Consider phase i of epoch ℓ of execution ξ for randomly chosen set of permutations Ψ. We reason about the probability of phase i belonging to one of the classes illustrated in Figure 4.3, and about the work that phase i contributes to the total work incurred in the execution, depending on its classification. From Lemma 4.22(a) we get that phase i may be a majority phase with probability at least $e^{-p\log p}$ which is a very small probability. More precisely, the probability that for a set of permutations Ψ, in execution ξ obtained for Ψ some phase i is a majority phase, is $O(\log^2 p \cdot e^{-p\log p}) = e^{-\Omega(p\log p)}$, 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 4.22(a)). We can not say anything about the probability of a minority phase to be a reliable or unreliable, since this depends on the specific execution. Note however, that by definition, we cannot have more than $O(\log p)$ unreliable minority phases in any execution ξ (at least one processor must remain operational). Moreover, the work incurred in an unreliable minority phase i of an epoch ℓ in any execution ξ is bounded by

$$O(p_i(\xi)\cdot(T_\ell+g_\ell)) = O\left(\frac{p}{2^{\ell-1}}\cdot\left(\frac{n+p\log^3 p}{\frac{p}{2^\ell}\log p} + \log^2 p\right)\right) = O\left(\frac{n}{\log p} + p\log^2 p\right).$$

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

From Lemmas 4.21 and 4.22(b) we get that a reliable minority phase may be fractional or optimal with high probability $1 - e^{-T_\ell/16}$, whereas it may be unproductive with very small probability $e^{-T_\ell/16} \leq e^{-\log^2 p}/16$. Using a similar argument as for majority phases, we get that for almost all sets of permutations Ψ (probability $1 - O(\log^2 p \cdot e^{-T_\ell/16}) \geq 1 - e^{-\Omega(T_\ell)}$) and for

every execution ξ, there is no minority reliable unproductive phase. The work incurred by a fractional phase i of an epoch ℓ in any execution ξ is bounded by $O(p_i(\xi) \cdot (T_\ell + g_\ell)) = O(\frac{n}{\log p} + p \log^2 p)$. Also note that by definition, there can be at most $O(\log_{3/4} n)$ $(= O(\log p)$ since $n \leq p^2)$ fractional phases in any execution ξ and hence, the total work incurred by all fractional reliable minority phases in any execution ξ is $O(n + p \log^3 p)$. We now consider the optimal reliable minority phases for any execution ξ. Here we have an optimal allocation of tasks to processors in $V_i(\xi)$. By definition of optimality, in average one task in $U_i(\xi) \setminus U_{i+1}(\xi)$ is performed by at most *four* processors from $V_{i+1}(\xi)$, and by definition of reliability, by at most *eight* processors in $V_i(\xi)$. Therefore, in optimal phases, each unit of work spent on performing a task results to a unique task completion (within a constant overhead), for any execution ξ. It therefore follows that the work incurred in all optimal reliable minority phases is bounded by $O(n)$ in any execution ξ.

Therefore, from the above we conclude that when $n \leq p^2$, for random set of permutations Ψ the work complexity of algorithm DOALL_ε executed on such set Ψ is $S = O(n + p \log^3 p)$ with probability $1 - e^{-\Omega(p \log p)} - e^{-\Omega(T_\ell)} = 1 - e^{-\Omega(T_\ell)}$ (the probability appears only from analysis of majority and unproductive reliable minority phases). Consequently such set Ψ exists. Also, from Lemma 4.22 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 $\text{GOSSIP}_{\varepsilon/3}$). 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 4.15). Using the result of Case 1 and selected set Ψ and constant β, we get that $S = O(n' + p \log^3 p) \cdot \Theta(n/p^2) = O(p^2 \cdot n/p^2 + n/p^2 \cdot p \log^3 p) = O(n)$. The message complexity is derived with the same way as in Case 1. □

4.5 Open Problems

As demonstrated by the gossip-based *Do-All* algorithm presented in this chapter, efficient algorithms can be designed that do not rely on single coordinators or reliable multicast to disseminate knowledge between processors. Gossiping seems to be a very promising alternative. An interesting open problem is to investigate whether a more efficient gossip algorithm can be developed that could yield an even more efficient *Do-All* algorithm.

An interesting problem is to perform a failure-sensitive analysis for the *iterative Do-All* problem using point-to-point messaging. Recall that if an algorithm solves the *Do-All*$_{\mathcal{A}_C}(n, p, f)$ problem with work $O(x)$ then this algorithm can be iteratively used to solve the *r-Do-All*$_{\mathcal{A}_C}(n, p, f)$ problem with work $r \cdot O(x)$. However, it should be possible to produce an improved upper

bound, for example, as we did in the previous chapter for the model with crashes and reliable multicast.

4.6 Chapter Notes

Dwork, Halpern, and Waarts [30] introduced and studied the *Do-All* in the message-passing model. They developed several deterministic algorithms that solved the problem for synchronous crash-prone processors. To evaluate the performance of their algorithms, they used the task-oriented work complexity W and the message complexity measure M. They also used the *effort* complexity measure, defined as the sum of W and M. This measure of efficiency makes sense for algorithms for which the work and message complexities are similar. However, this makes it difficult to compare relative efficiency of algorithms that exhibit varying trade-offs between the work and the communication efficiencies.

The first algorithm presented in [30], called protocol $\mathcal{B}$ has effort $O(n + p\sqrt{p})$, with work contributing the cost $O(n + p)$ and the message complexity contributing the cost $O(p\sqrt{p})$ toward the effort. The running time of the algorithm is $O(n + p)$. The algorithm uses synchrony to detect processor crashes by means of timeouts. The algorithm operates as follows. The n tasks are divided into chunks and each chunk is divided into sub-chunks. Processors checkpoint their progress by multicasting the completion information to subsets of processors after performing a subchunk, and broadcasting to all processors after completing chunks of work. Another algorithm, called protocol $\mathcal{C}$ has effort $O(n + p \log p)$. It has optimal work $W = O(n + p)$, message complexity $M = O(p \log p)$ and time $O(p^2(n+p)2^{n+p})$. This shows that reducing the message complexity may cause a significant increase in time. Protocol $\mathcal{D}$ is another *Do-All* algorithm that obtains work optimality and it is designed for maximum speed-up, which is achieved with a more aggressive check-pointing strategy, thus trading-off time for messages. The message complexity is quadratic in p for the fault-free case, and in the presence of $f < p$ crashes the message complexity degrades to $\Theta(fp^2)$.

De Prisco, Mayer, and Yung [25] provided an algorithmic solution for *Do-All* considering the same setting as Dwork *et al.*, (synchrony, processor crashes) but using the total-work (available processor steps) complexity measure S. They use a "lexicographic" criterion: first evaluate an algorithm according to its total-work and then according to its message complexity. This approach assumes that optimization of work is more important than optimization of communication. They present a deterministic algorithm, call it DMY, that has $S = O(n + (f + 1)p)$ and $M = O((f + 1)p)$. The algorithm operates as follows. At each step all the processors have a consistent (over)estimate of the set of all the available processors (using checkpoints). One processor is designated to be the coordinator. The coordinator allocates the undone tasks according to a certain load balancing rule and waits for notifications

of the tasks which have been performed. The coordinator changes over time. To avoid a quadratic upper bound for S, substantial processor slackness is assumed ($p \ll n$).

The authors in [25] also formally show a lower bound of $S = \Omega(n + (f+1)p)$ for any algorithm using the stage-checkpoint strategy, this bound being quadratic in p for f comparable with p. Moreover, any protocol with at most one active coordinator (that is, a protocol that uses a single coordinator paradigm) is bound to have $S = \Omega(n+(f+1)p)$. Namely, consider the following behavior of the adversary: while there is more than one operational processor, the adversary stops each coordinator immediately after it becomes one and before it sends any messages. This creates pauses of $\Omega(1)$ steps, giving the $\Omega((f+1)p)$ part. Eventually there remains only one processor which has to perform all the tasks, because it has never received any messages, this gives the remaining $\Omega(n)$ part. Algorithm AN (presented in Chapter 3) beats this lower bound by using a multicoordinator approach; however it makes use of reliable multicast. Algorithm DoAll$_\varepsilon$ presented in this chapter beats this lower bound by neither using checkpointing nor single-coordinators paradigms; instead it uses a gossip algorithm for the dissemination of information.

Galil, Mayer, and Yung [38], while working in the context of Byzantine agreement [78] assuming synchronous crash-prone processors, developed an efficient algorithm, call it GMY, that has the same total-work bound as algorithm DMY ($S = O(n + (f+1)p)$) but has better message complexity: $M = O(fp^\varepsilon + \min\{f+1, \log p\}p)$, for any $\varepsilon > 0$. The improvement on the message complexity is mainly due to the improvement of the checkpoint strategy used by algorithm DMY by replacing the "rotating coordinator" approach with what they called the "rotating tree" (*diffusion tree*) approach.

Chlebus, Gasieniec, Kowalski, and Shvartsman [16] developed a deterministic algorithm that solves *Do-All* for synchronous crash-prone processors with combined total-work and message complexity $S + M = O(n + p^{1.77})$. This is the first algorithm that achieves subquadratic in p combined S and M for the *Do-All* problem for synchronous crash-prone processors. They present another deterministic algorithm that has total-work $S = O(n + p\log^2 p)$ against f-bounded adversaries such that $p - f = \Omega(p^\alpha)$ for a constant $0 < \alpha < 1$. They also show how to achieve $S + M = O(n + p\log^2 p)$ against a linearly-bounded adversary by carrying out communication on an underlying constant-degree network.

The presentation in this chapter is based on a paper by Georgiou, Kowalski, and Shvartsman [44]. The proofs of Lemmas 4.4, 4.5 and Theorem 4.6 appear there. For the probabilistic method and its applications see the book of Alon and Spencer [4]. The notion of the *left-to-right maximum* is due to Knuth [71] (p. 13).

The complexity results presented in this chapter involve the use of conceptual communication graphs and sets of permutations with specific combinatorial properties. Kowalski, Musial, and Shvartsman [75] showed that such combinatorial structures can be constructed efficiently.

Additionally, observe that the complexity bounds do not show how work and message complexities 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. Georgiou, Kowalski, and Shvartsman show how this can be achieved in [44]. The main idea relies on the fact that checkpointing is rather efficient for a small number of failures. So, the authors use algorithm DOALL$_\varepsilon$ in conjunction with the check-pointing algorithm DMY [25], where the check-pointing and the synchronization procedures are taken from algorithm GMY [38]; in addition they use a modified version of algorithm GOSSIP$_\varepsilon$, optimized for a small number of failures. The resulting algorithm achieves total work $S = O(n + p \cdot \min\{f + 1, \log^3 p\})$ and message complexity $M = O(fp^\varepsilon + p \min\{f + 1, \log p\})$, for any $\varepsilon > 0$. More details can be found in [44].

Chlebus and Kowalski [18] were the first to define and study the *Gossip* problem for synchronous message-passing processors under an adaptive adversary that causes processor crashes (this is the version of the *Gossip* problem considered in this chapter); they developed an efficient gossip algorithm and they used it as a building block to obtain an efficient synchronous algorithm for the consensus problem with crashes. In a later work [19], the same authors developed another algorithm for the synchronous *Gossip* problem with crashes and used it to obtain an efficient early-stopping consensus algorithm for the same setting. More details on work on gossip in fault-prone distributed message-passing systems can be found in the survey of Pelc [96] and the book of Hromkovic, Klasing, Pelc, Ruzicka, and Unger [59].