WARSAW UNIVERSITY

FACULTY OF MATHEMATICS,
INFORMATICS AND MECHANICS

DISTRIBUTED ALGORITHMS TO PERFORM INDEPENDENT TASKS IN NETWORKS WITH PROCESSOR FAULTS

Ph.D. Thesis

Dariusz R. Kowalski

Thesis supervisor:
dr hab. Bogdan S. Chlebus

Warsaw, May 2001

Acknowledgement

Thanks are due to my thesis supervisor Bogdan Chlebus for his guidance and inspiration.

This dissertation contains results obtained as a joint work with Bogdan Chlebus, Leszek Gąsieniec, Andrzej Lingas and Alex Shvartsman. I thank them all for a rewarding and fruitful cooperation.

The utmost thanks go to my wife for her constant support and patience.

D.R.K.

Contents

Chapter 1

Introduction

One of fundamental problems in distributed computing is how to perform a number of independent tasks by a system in which processors are prone to failures. We consider this problem when synchronous processors communicate by exchanging messages and may fail by crashing. Tasks that we consider are similar, independent and idempotent. This problem is called *Do-All*.

Protocols performing independent tasks in networks prone to failures have been applied in practice, among specific implementations we mention Piranha [35, 11], Condor [45], Amber [24] and the V-system [56]. Typical tasks in such systems are:

- checking large space of solutions, like a validity of assignment to a given Boolean formula;
- generating independent random trials, used for example in verification of polynomial identities or in primality testing;
- independent calculations, like sieving operations during integer factorization.

In our work we assume that processors operate synchronously, with a global clock controlling their steps. We consider two kinds of distributed systems, the difference is in the way messages are exchanged:

- a general *message-passing system*: a processor can multicast a message to any subset of processors in a time step;
- *multiple-access channel*: a processor can broadcast a message to all the processors in one step, but simultaneous broadcasts block the channel.

We develop algorithms which are robust in that they perform all the tasks even if only one processor remains available, but are simultaneously optimized against specific adversaries controlling occurrence of faults.

A processor does not need to perform a task itself if it learns that someone else has already done so. This may justify treating work and communication as comparable resources when measuring the performance of algorithms. We measure work as the total number of steps of computation. Communication is measured as the number of point-to-point messages.

In the remaining part of this section we use terminology which is precisely defined in Chapter 2. The number of tasks is denoted by t and the number of processors by p. If the adversarial model under consideration has an upper bound on the number of faults then this number is denoted by f.

1.1 Previous work

The problem *Do-All* was introduced by Dwork, Halpern, and Waarts [21], and next investigated in a number of papers [13, 16, 17, 20, 29]. Popular algorithmic paradigms that were employed included balancing work and checkpointing the progress made. Certain processors might be designated as coordinators to collect and disseminate information. The most popular model of failures studied was that of fail-stop ones. The primary measures of efficiency of algorithms used in [21] were task-oriented work, in which each performance of a task contributes a unit, and communication, measured as the number of point-to-point messages. This paper also proposed effort as a measure of performance, which is work and communication combined; one algorithm presented in [21] has effort $\mathcal{O}(t + p\sqrt{p})$. Early work has concentrated on the adversary who could fail all the processors but one. Recent work has usually concerned optimizing solutions against weaker adversaries, while preserving correctness in the worst-case scenario of arbitrary failure patterns, which guarantee only at least one available processor.

De Prisco, Mayer, and Yung [20] were the first to use the available processor steps as a measure of work. This measure counts all the steps of computation of non-faulty processors until all the tasks have been done. They present an algorithm which has work $\mathcal{O}(t + (f+1)p)$ and message complexity $\mathcal{O}((f+1)p)$. Galil, Mayer and Yung [29] improved the message complexity to $\mathcal{O}(fp^{\varepsilon} + \min\{f+1, \log p\}p)$, for any positive ε, while maintaining the same work. This was achieved as a by-product of their work on Byzantine agreement with stop-failures, for which they found a message-optimal solution.

Chlebus, De Prisco and Shvartsman [13] developed algorithms based on aggressive coordination, with the number of coordinators growing exponentially during the periods of complete coordinator failures. Their algorithms rely on the property of multicast to be atomic: if a processor fails during a multicast then either none or all of the messages are delivered to the recipients. One of their algorithms achieves $\mathcal{O}((t+p\log p/\log\log p)\log f)$ available processor steps with the communication bounded by $\mathcal{O}(t+p\log p/\log\log p+fp)$. Another algorithm is designed to handle restarted processors. Note that straightforward ways to integrate restarts into a useful computation may have a cost, in terms of the available processor steps, well bigger than the speedup obtained in terms of the task-oriented work. It is the only algorithm known that incorporates restarts efficiently, it uses $\mathcal{O}((t+p\log p+f)\cdot\min\{\log p, \log f\})$ available processor steps and its message complexity is $\mathcal{O}(t+p\log p+fp)$.

Solutions to the problem *Write-All* in the model with memory snapshots are closely related to solutions for *Do-All* with complete information of processors in each step (problem *Write-All* is defined and discussed in Section 1.3). There is a natural adaptation of both lower bounds and algorithms solving the problem *Write-All* in the model with memory snapshots to the problem *Do-All*. Kanellakis and Shvartsman [37] considered *Write-All* in this model and proved a lower bound $\Omega(n\log n/\log\log n)$ on the available processor steps. They also developed an algorithm with a matching work performance. This algorithm requires communication $\mathcal{O}(t+p^2\log t/\log\log t)$ if adapted to solve *Do-All*. We summarize the performance bounds of algorithms solving the problem *Do-All* prior to our research in Table 1.1.

	papers [20, 29]	paper [13]	paper [37]
work	$\mathcal{O}(t+(f+1)p)$	$\mathcal{O}(t+\frac{p\log p}{\log\log p}\log f)$	$\mathcal{O}(t+\frac{p\log t}{\log\log t})$
messages	$\mathcal{O}(fp^{\varepsilon}+\min\{f+1,\log p\}p)$	$\mathcal{O}(t+\frac{p\log p}{\log\log p}+fp)$	$\mathcal{O}(t+\frac{p^2\log t}{\log\log t})$

Table 1.1: Performance of previous algorithms against f-bounded adversary.

Our algorithms are optimized against adversaries who have to leave operational at least a constant fraction of processors. Table 1.2 gives performance bounds of algorithms known in the literature if run against such restricted adversaries.

	papers [20, 29]	paper [13]	paper [37]
work	$\mathcal{O}(t+p^2)$	$\mathcal{O}(t+\frac{p\log^2 p}{\log\log p})$	$\mathcal{O}(t+\frac{p\log t}{\log\log t})$
messages	$\mathcal{O}(p^{1+\varepsilon})$	$\mathcal{O}(t+p^2)$	$\mathcal{O}(t+\frac{p^2\log t}{\log\log t})$

Table 1.2: Performance of previous algorithms against linearly bounded adversaries.

As we shall see, all our algorithms for the message-passing model achieve a better performance against such adversaries. At this point we need a word of caution: one should not compare algorithmic solutions for the problem *Do-All* based *solely* on the performance bounds as given in these two tables. This is because algorithms are usually designed and optimized for specific distributed environments and adversarial models.

The most popular adversarial model in previous research on the *Do-All* problem was that of UNBOUNDED adversary, it was used in [13, 20, 21, 29]. The only exceptions were the paper [13], where an adversarial model allowing restarts was considered.

1.2 Our results

We develop algorithms that solve the *Do-All* problem against the adversary UNBOUNDED, but their performance is optimized against linearly-bounded adversaries. This means that we prove algorithms to be correct in the former adversarial model but their performance is analyzed in the latter.

Our work can be partitioned into the following three parts:

(1) WEAKLY-ADAPTIVE LINEARLY-BOUNDED adversary and the message-passing model;

(2) STRONGLY-ADAPTIVE LINEARLY-BOUNDED adversary and the message-passing model;

(3) f-BOUNDED adversaries and the broadcast-network model.

We discuss them briefly in what follows.

Message-passing and weakly-adaptive linearly-bounded adversaries

In this setting we assume atomic communication. We develop a randomized algorithm and prove a lower bound.

The randomized algorithm has expected work and communication $\mathcal{O}(t + p \cdot (1 + \log^* p - \log^*(p/t)))$. (Function $\log^*$ is defined as the number of iterations of the log function which is required to go with the value of the function down to 1.)

We prove a lower bound $\Omega(t + p \cdot \log t / \log\log t)$ on the expected work for algorithms solving *Do-All* against STRONGLY-ADAPTIVE LINEARLY-BOUNDED adversary. This result is related to a lower bound about *Write-All* with memory snapshots, for the adversary who can fail all but one processors (see [37]). We show that the lower bound on work $\Omega(t + p \cdot \log t / \log\log t)$ is matched by the work performed by a simple deterministic algorithm. This algorithm is closely related to algorithm W_{snap} developed in [37], which solves *Write-All* in the model with memory snapshots. The communication complexity of our algorithm is $\mathcal{O}(t + p^2 \log t / \log\log t)$.

If $t = p$ then the randomized algorithm has expected work $\mathcal{O}(p \cdot \log^* p)$ against Weakly-Adaptive Linearly-Bounded, but any algorithm against the adversary Strongly-Adaptive Linearly-Bounded requires expected work $\Omega(p \log p / \log\log p)$. There are two conclusions from the performance of the algorithm and the lower bound. The first one is that Strongly-Adaptive Linearly-Bounded adversary is more powerful than Weakly-Adaptive Linearly-Bounded one against randomized algorithms. Note that, for deterministic algorithms, there is no difference in power between strongly and weakly adaptive adversaries. This gives the second conclusion: randomized algorithms against Weakly-Adaptive Linearly-Bounded adversary can be more efficient than deterministic ones in terms of expected work and communication. In other words: randomization helps against Weakly-Adaptive Linearly-Bounded adversary.

These results are described in Chapter 3 and appeared in a preliminary form in [17].

Message-passing and strongly-adaptive linearly-bounded adversaries

In this setting we do not assume atomic communication. We develop three algorithms for the *Do-All* problem in this model. We start with a deterministic constructive algorithm, then trade either determinism or constructiveness for better work and communication performance.

The first constructive deterministic algorithm is based on load-balancing and solves the *Do-All* problem with work and communication $\mathcal{O}(t + p \cdot \sqrt{t+p} \cdot \log^{3/2} p)$. Then we develop a randomized algorithm which solves the *Do-All* problem with work and communication $\mathcal{O}(t + p \cdot \log^2 p / \log\log p)$ with high probability. Finally we show that there is a deterministic algorithm which solves the *Do-All* problem with work and communication $\mathcal{O}(t + p \cdot \log^2 p)$. The algorithm uses a family of permutations of tasks which determine the order in which processors perform tasks. The algorithm is not constructive in the sense that the families of permutations with the required good properties are proved to exist by the probabilistic method.

If $t = p$ then the first algorithm has the work and communication $\mathcal{O}((p \log p)^{3/2})$. The second one has the expected work and communication $\mathcal{O}(p \log^2 p / \log\log p)$, which is $\mathcal{O}(p \cdot \text{polylog}(p))$ and hence substantially better. The third (deterministic) algorithm has the work and communication $\mathcal{O}(p \cdot \log^2 p)$, and is the first known deterministic algorithms achieving work performance close to linear one, modulo a polylogarithmic factor, in a situation when an adversary may fail a linear fraction of processors. Each of these three algorithms is an improvement over the algorithms known before, since all of those algorithms have the worst case work and communication $\Omega(p^2)$ in the considered model of Linearly-Bounded adversaries. The performance of the third most efficient algorithm is close to the lower bound

$\Omega(p \log p / \log\log p)$ on work proved in Chapter 3. An interesting feature of our algorithms is that they do not use such techniques as coordinators or checkpointing, instead processors spread information locally over certain communication graph.

These results are described in Chapter 4 and have been announced in a preliminary form in [16].

Broadcast networks and f-bounded adversaries

We consider the *Do-All* problem in the communication setting of a broadcast network. More precisely, the communication is over a multiple-access channel, either with or without collision detection.

First we show that the minimum amount $\Omega(t + p\sqrt{t})$ of work has always to be performed by any algorithm solving *Do-All*. This is an absolute lower bound on work performance of any algorithm, which does not depend on the availability of collision detection, randomization or the power of an adversary. We show that in a channel with collision detection this bound can be attained by a deterministic algorithm against any size-bounded adversary. The situation is more complex in a weaker channel without collision detection. We develop a deterministic algorithm for this channel which performs work $\mathcal{O}(t + p\sqrt{t} + p \cdot \min\{f, t\})$ against adversary who can fail at most f processors, even if the number of failures is the only restriction on the power of the adversary. This is shown to be optimal by a matching lower bound. Now it could happen that if we wanted to optimize our solutions against weaker adversaries then part $\mathcal{O}(p \cdot \min\{f, t\})$ in the performance bound could be decreased. This indeed is the case: we show that a randomized algorithm can have expected minimum work $\Theta(t + p\sqrt{t})$ against certain weakly adaptive adversaries. The conclusion is that randomization helps if collision detection is not available and the adversary is sufficiently restricted. The maximum number of faults when this phenomenon happens is a constant fraction of the number of all the processors. Next we show a lower bound which implies that if only $t = o(p^2)$ then a weakly-adaptive f-bounded adversary with $f = p \cdot (1 - o(1/\sqrt{t}))$ can force any algorithm for the channel without collision detection to perform asymptotically more than the minimum work $\Theta(t + p\sqrt{t})$. All our algorithms solving *Do-All* in multiple-access channel require only $\mathcal{O}(\log p)$ bits for one message and sometimes only $\mathcal{O}(1)$.

These results are in Chapter 5 and have been announced in a preliminary form in [18].

The problem *Do-All* has been considered before only for the general message-passing system, and all the solutions presented prior to our work were deterministic. Because of that our randomized solutions and also algorithms for the multiple-access channel cannot be directly compared to previous work.

1.3 Related work

Distributed computing in general and distributed algorithms in particular are mature areas of research in computer science. Comprehensive accounts of results regarding fundamental algorithmic problems can be found in books of Attiya and Welch [7], Lynch [46] and Tel [55]. Fich and Ruppert [25] present a survey of lower bounds for problems in fault tolerant distributed computing.

Problem Write-All

The problem *Do-All* specialized to shared-memory models has been investigated before, especially the variant called *Write-All* in which the operation of reading/writing from a memory cell/register is considered to be an individual task. In other words, the goal is to use p failure- or delay- prone processors to update t shared memory locations. A solution to this problem can be used iteratively to simulate computations of a general-purpose shared-memory fully operational computer on one prone to processor faults. The problem *Write-All* was introduced by Kanellakis and Shvartsman [37], for a comprehensive overview see [38]. Algorithms for the *Write-All* problem have been developed in a series of papers, including those by Anderson and Woll [5], Kedem, Palem, Rabin and Raghunathan [39], Kedem, Palem and Spirakis [41], and Martel, Park and Subramonian [49]. To have concise performance bounds, let n denote both the number of processors and the size of the array. The best known deterministic algorithm performs work $\mathcal{O}(n \log^2 n/ \log\log n)$ against fail-stop failures, and $\mathcal{O}(n^{1+\epsilon})$, for arbitrary $\epsilon > 0$, if restarts may happen. Randomized solutions can attain the expected work $\mathcal{O}(n \log n)$ against failures with restarts. A logarithmic lower bound on time to solve *Write-All* deterministically was derived by Kedem, Palem, Raghunathan and Spirakis [40], and on expected time of randomized executions by Martel and Subramonian [50]. Kanellakis and Shvartsman [37] proved a lower bound $\Omega(n \log n/ \log\log n)$ on available processor steps in the model with memory snapshots, and developed an algorithm with a matching performance, where number n denotes both the number of processors and the size of the array. Buss, Kanellakis, Ragde and Shvartsman [10] proved a lower bound $\Omega(n \log n)$ on the available processor steps for any deterministic executions controlled by an adversary that can cause both failures and restarts, and developed an algorithm of a matching performance in the model with memory snapshots. See also [14, 27] for results of a recent more practically oriented research.

Multiple-access channel

The multiple-access channel is a special broadcast network ([8, 54]). It may be also interpreted as a single-hop radio network, especially in the context of the relevance of collision detection ([12]). Most of the previous research on the multiple-access channel has been concerned with methods of handling packets which the stations keep receiving and which need to be broadcast on the channel as soon as possible. The packets may be generated dynamically in a possibly irregular way which results in a bursty traffic. Techniques like time-division multiplexing are not efficient then, and a better throughput can be achieved if the control is distributed among the stations. This is done by conflict-resolution protocols, which arbitrate among the stations competing for access to the channel; among the most popular protocols is Aloha and the exponential backoff. If packets are generated dynamically then the basic problem is to have stable protocols, which do not make the channel clogged eventually. Recent work in that direction includes the papers by Goldberg, MacKenzie, Paterson and Srinivasan [28], Håstad, Leighton and Rogoff [33], and Raghavan and Upfal [52]; see also the survey of Gallager [26] for an account of the early research and that of Chlebus [12] for recent developments.

Static problems for a multiple-access channel concern a scenario when input data are allocated at the stations prior to the start of an algorithm. The problem of *selection* concerns the situation when some of the stations hold messages, the goal is to broadcast just any single message successfully. Willard [58] developed protocols solving the problem in the expected time $\mathcal{O}(\log\log n)$ in the channel with collision detection. Kushilevitz and Mansour [43] proved a lower bound $\Omega(\log n)$ for the selection problem if collision detection is not available, which yields an exponential gap between two models for this problem. A related problem of finding maximum among the keys stored in a subset of stations was considered by Martel [48].

There is a related *all-broadcast* problem in a multiple-access channel, in which a subset of k among n stations have messages, all of them need to be sent to the channel successively as soon as possible. Komlós and Greenberg [36] showed how to solve it deterministically in time $\mathcal{O}(k + k\log(n/k))$, where both numbers n and k are known. A lower bound $\Omega(k(\log n)/(\log k))$ was proved by Greenberg and Winograd [32].

Gąsieniec, Pelc and Peleg [30] compared various models of synchrony in multiple-access channel in the context of the *wakeup* problem, in which the system is started and the time when each station joins is controlled by an adversary, while the goal is to perform a successful broadcast as soon as possible. If the stations have access to a global clock then a wakeup can be realized in the expected time $\mathcal{O}(\log n)$ by a randomized algorithm. If the local clocks are not synchronized, there is a randomized solution working in the expected time $\mathcal{O}(n)$. It was also shown in [30] that deterministic algorithms require time $\Omega(n)$, and that there are deterministic schedules working in time $\mathcal{O}(n\log^2 n)$.

Adversarial models

Linearly-bounded adversaries have been used in other contexts in a number of papers, the following is a sample. Chlebus, Gambin and Indyk [15] considered CRCW PRAM with faulty memory cells, the occurrence of faults controlled by the Weakly-Adaptive Linearly-Bounded adversary. Diks and Pelc [23] studied deterministic broadcasting and fault diagnosis in a (complete) network with processor faults controlled by a Linearly-Bounded adversary. Kontogiannis, Pantziou, Spirakis and Yung [42] studied the fault prone BSP model of a parallel computer, with processor faults controlled by a certain stochastic linearly-bounded adversary.

Sparse communication networks

We mention some related research concerning problems in networks of bounded degree when up to $\mathcal{O}(n)$ processors may fail. Alon and Chung [2] showed that the expanders introduced in [44] and [48] have the following property: if a fraction of arbitrary nodes or edges are removed then there is a path of linear size, what allows to simulate arrays. Upfal [57] showed how an almost everywhere agreement can be achieved with these many faults, extending the result from [22]. To this end, he used the same family of expanders and showed that a linear-sized component of a logarithmic diameter remains if some fraction of arbitrary nodes is removed.

Other related problems

Certain specific version of the problem of performing independent tasks on an asynchronous system was considered by Bridgland and Watro [9]. They showed that this problem is not always solvable in the model considered. The authors provided tight bounds on the number of failures which guarantee solvability.

There is another related problem called *Collect*, it was introduced by Saks, Shavit and Woll [53]. It is about a number of processors, each containing a value in a shared memory register: the goal the processors need to achieve is to learn all the values. A processor increases its knowledge in a step by reading a register: then it can add the read value to the contents of its own register. The number of read/write operations is the measure of performance. There is an adversary who controls timing of the processors in an asynchronous computation. Ajtai, Aspnes, Dwork and Waarts [1] showed that the problem for n processors can be solved deterministically with work $\mathcal{O}(n^{3/2}\log n)$, by an adaptation of the algorithm of Anderson and Woll [5]. Aspnes and Hurwood [6] developed a randomized algorithm achieving work $\mathcal{O}(n\log^3 n)$ with high probability. A lower bound $\Omega(n\log n)$ for this problem was given in [53].

The *Do-All* problem is related to Byzantine Agreement and Gossiping problems, see [21] and [29]. Our algorithms solving the *Do-All* problem in the message-passing model have the following property: each processor deciding to halt has performed certain tasks itself or received messages confirming completion of the remaining tasks. If we consider the ith task as learning the value held by the ith processor then each processor can learn the values of all the operational processors if the system runs one of our algorithms solving the *Do-All* problem.

1.4 Contents of the dissertation

In Chapter 2 we describe in detail the models of a message-passing system and a multiple-access channel. We also specify the *Do-All* problem precisely. The models of faults which can happen in the system are categorized by the respective adversarial models. Finally we define complexity measures.

In Chapter 3 we study solutions of the *Do-All* problem against linearly-bounded adversaries in a message-passing model with atomic communication. We develop a randomized algorithm and prove a lower bound.

In Chapter 4 we study solutions of the *Do-All* problem against the STRONGLY-ADAPTIVE LINEARLY-BOUNDED adversary in the message message-passing model. We present a general scheme of algorithms and three possible implementations: deterministic algorithm based on load-balancing of tasks, a randomized algorithm, and finally a non-constructive deterministic algorithm.

In Chapter 5 we explore the problem *Do-All* in the setting of a multiple-access channel. We develop efficient deterministic algorithms for both the channels with and without collision detection. We also prove matching lower bounds. Next we develop a randomized algorithm for the channel without collision detection, and show it to be provably better than deterministic algorithms in some adversarial models.

The final Chapter 6 contains conclusions.

Chapter 2

Technical preliminaries

2.1 General setting

There are p processors, each with integer *unique identifier* (UID) in the interval $[1..p]$. The processor with UID equal to v is denoted by $\mathcal{P}_v$. Each processor knows its UID and the number p.

A computation of each processor proceeds through a sequence of states, during which it can perform certain operations. Each processor has a *start* and a *halt* state. A processor which is in a halt state is called *halted*. We do not describe states formally, they are determined implicitly by the contents of local variables.

Processors are prone to fail-stop failures.

Computations are synchronous and controlled by a global clock. A clock cycle is called a *step*. We consider two kinds of steps:

communication: generating and sending messages;

local computation: receiving messages and transforming the state.

Typically a number of steps are grouped together into *rounds*, each containing communication and local-computation steps, which together perform a logical part of algorithm.

2.2 The *Do-All* problem

There are t tasks. The processors need to perform all of them. Each task has a unique identifier in the interval $[1..t]$. Each processor knows number t and can perform any task, given its identifier, during one step of computation.

The following three properties of tasks are assumed:

similarity: each task takes one step to perform;

independence: tasks may be performed in any order;

idempotence: each task may be performed many times and concurrently.

An instance of the problem *Do-All* is completed when all the tasks are done and when every operational processors has halted. An algorithm solving the *Do-All* problem is said to be *reliable* if the following two conditions are satisfied:

Correctness: all the tasks are eventually performed, if at least one processor remains non-faulty;

Termination: all the processors eventually halt, unless failed.

2.3 Models of communication

We consider two underlying models of communication: the message-passing model and the multiple-access channel.

Message-passing system

In the *message-passing* model processors communicate by exchanging messages. Each processor can multicast messages to any subset of processors during one sending operation.

There is no upper bound on the size of messages. A message sent is delivered to its recipients by the begining of the next step. We assume that links among the processors are reliable, in that no messages are lost or corrupted in transit.

We say that *communication is atomic* if each operation of a multicast by a processor either delivers the messages to all the recipients or to none in case of a processor failure during performing the multicast.

Multiple-access channel

This model is a kind of broadcast network, which is related to single-hop radio networks. That is why a special terminology is used in this context. The processors in this setting are usually called *stations*. We say that a station can *hear* the information it receives from the channel. We assume that a message sent on the channel is delivered to all the stations, but if many are sent simultaneously then they interfere with each other and are received as garbled. If a message sent on the channel is heard by a station then the message is said to be *successfully delivered* to it. A sufficient size of a packet to carry a single message in our algorithms is $\mathcal{O}(\log p)$ bits.

The *multiple-access channel* operates according to the following rules. If exactly one station performs a broadcast in a communication step then the message is successfully delivered to all the stations. If more of them broadcast simultaneously in a step then mutual collisions happen and no station successfully receives any of these messages.

It is convenient to use the notion of *noise* in this context, which should not be confused with a form of failure sometimes considered in models of communication. If no messages are sent then we say that the stations receive only *background noise*, which is distinct from any meaningful message. If more than one messages are broadcast simultaneously in a step then a *collision* happens, and no station can hear any of these messages. We consider two models depending on what feedback the stations receive if a collision happens:

channel without collision detection: each station can hear the background noise.

channel with collision detection: each station can hear the *interference noise*, which is distinct from the background noise.

If the stations attached to the channel do not receive a meaningful message at a step then there are two possible reasons: either none or more than one messages were sent. If the ability to distinguish between these two cases by collision detection is available then the channel is also called to be *with a ternary feedback*, because of the three possible events on the channel recorded by the attached stations:

0 : no messages sent,

1 : a meaningful message received,

2+ : a collision signal received.

The numbers in parenthesis denote the number of simultaneous broadcasts.

2.4 Models of failures

Failures are dynamic and occur in the course of executing an algorithm. They may happen at any time, in particular while performing a communication step or a local-computation step. Processors fail by crashing. If a processor crashes then it stops any activity and never restarts. A processor which has not failed yet at a step/round is said to be *operational* in this step/round. We distinguish *active* processors: these are operational processors that are not in a halt state.

Failures are generated by *adversaries* which are personifications of external forces and conditions affecting patterns of occurrence of failures. An adversary knows the algorithm against which it competes. Halts do not restrict an adversary since halted processors are not considered to be faulty. We consider only failure patterns in which at least one processor remains always operational, otherwise there is no solution to the *Do-All* problem. Links are assumed to be reliable.

An adversary is *size-bounded* if it may fail at most f processors, for a parameter $0 \leq f < p$. We may refer to a size-bounded adversary as f*-bounded* to make the value of parameter f explicit. If f is a constant fraction of p then we denote it by $(1-c)\cdot p$, where $0 < 1-c < 1$. The respective size bounded adversary adversary is then called a *linearly-bounded with constant* $1-c$, or simply *linearly bounded*. In this case at least cp processors remain operational until the end of computation.

The property of an adversary to be allowed to make decisions online is its *adaptiveness*, it usually is restricted by imposed conditions. An adversary is said to be *adaptive with condition* $\mathcal{C}$ if it may make its decisions online, the only constraint being that the condition $\mathcal{C}$ has to be satisfied in each execution.

We consider the following specific adversaries:

Strongly-Adaptive f-Bounded: is adaptive with the condition that at most f processors are failed, where $0 \leq f < p$.

Weakly-Adaptive f-Bounded: it is adaptive with the following condition, where $0 \leq f < p$:

1) It needs to select f *failure-prone* processors prior to the start of an algorithm;
2) It may fail only the selected failure-prone processors in the course of an algorithm.

We write simply f-Bounded for Strongly-Adaptive f-Bounded.

If f is a constant fraction of p then Strongly-Adaptive f-Bounded is called Strongly-Adaptive Linearly-Bounded or simply Linearly-Bounded.

If f is a constant fraction of p then Weakly-Adaptive f-Bounded is called Weakly-Adaptive Linearly-Bounded.

The adversary Unbounded is the same as (Strongly-Adaptive) $(p-1)$-Bounded.

We require algorithms to be correct against any strategy of the adversary Unbounded, this property has been formally defined in Section 2.2 as reliability.

Note that there is no difference in power between strongly and weakly-adaptive size-bounded adversaries when they compete against deterministic algorithms: an adversary knows the future behavior of processors and may compare all the possible scenarios of failures and decide in advance which processors to fail at which steps.

2.5 Complexity measures

We consider the following performance measures of algorithms: work and communication.

Work

In our definition *work* counts the total number of steps of computation of active processors, including idling. A processor may seize to contribute to work in two ways. The first one is voluntary, when it withdraws from the computation towards completing the given instance of the problem by halting. The other one is forced and happens when the processor is failed by an adversary.

Work as defined in this way is usually called the *available processor steps*. It was first defined and studied by Kanellakis and Shvartsman (see [37]) in the context of PRAM computations.

Each processor may increases the work by the very fact that it exists and is available. Our definition of work allows processors to switch to other algorithms without the necessity to reach agreement about termination or even before all the tasks have been completed.

Yet another alternative is a *task-oriented work* in which one counts the number of performed tasks including multiplicities, that is, each performance of a task contributes a unit to the measure. This definition of work was assumed in [21]. Such work allows algorithms to communicate by timeouts for free, which can be used to have algorithms that are simultaneously work efficient and exponential in time.

Communication

The measure of *communication* counts the number of point-to-point messages sent over the network. More precisely, each transfer of information from one processor to another during a communication step contributes a unit, regardless of the size of the data transferred. We use this measure only in the message-passing system.

Note that in principle no messages are necessary since each operational processor could perform all the tasks itself. Then the work performed would need to be of order tp, which is usually prohibitively large and is the reason why algorithms try to trade communication for work. Often work and communication could be considered as comparable resources. To capture this one can consider *effort* as a measure of performance, which is defined to be the sum of work and communication; it was proposed in [21].

Remarks

If a processor halts then it is not to be interpreted that it chooses to be idle, but rather that it disengages itself from the current instance of the problem and makes itself available to the system to perform other jobs.

Our definition of work or communication is such that they count only contributions accrued while performing tasks. They do not include pre-processing needed to distribute the input among the processors and set them to work, nor post-processing to prepare the system to be fully available for other purposes.

Chapter 3

Weakly adaptive adversaries

In this chapter we consider the problem *Do-All* in the following setting:

Model of communication: message-passing

Multicast: atomic

Adversaries: weakly- and strongly-adaptive linearly bounded

We emphasize the property 'weakly-adaptive' in the title because the main contribution of this chater is an algorithm which is efficient against the adversary WEAKLY-ADAPTIVE LINEARLY-BOUNDED.

Let $\log^*$ be the number of iterations of the log function required to go with the value of function down to 1. Formally: $\log^{(1)} x = \log x$, $\log^{(k+1)} x = \log(\log^{(k)} x)$, and $\log^* x = \min_k[\log^{(k)} x \leq 1]$.

We develop a randomized algorithm RA attaining both the expected work and communication $\mathcal{O}(t + p \cdot (1 + \log^* p - \log^*(p/t)))$ against any WEAKLY-ADAPTIVE LINEARLY-BOUNDED adversary. This result is in contrast with the performance of algorithms against strongly adaptive adversaries, which has to be $\Omega(t + p \log t / \log \log t)$ in terms of work, as shown in Section 3.3. This implies that STRONGLY-ADAPTIVE LINEARLY-BOUNDED adversary is more powerful than WEAKLY-ADAPTIVE LINEARLY-BOUNDED one. This also shows that randomized solutions of the problem *Do-All* are more efficient than deterministic scheduling strategies against WEAKLY-ADAPTIVE LINEARLY-BOUNDED adversaries.

Finally we prove that this lower bound is matched by the performance of a deterministic algorithm.

3.1 Randomized algorithm RA

In this section we develop a randomized algorithm RA solving *Do-All* and optimized against any WEAKLY-ADAPTIVE LINEARLY-BOUNDED adversary. It is based on random coordination and distributed random allocation paradigms.

Coordinators and workers

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

Phases

The computation proceeds in *phases* comprised of four consecutive rounds. A phase may be performed in one of two modes: work or election. *Work* phase consists of the following four rounds:

Round 1: performing a task,

Round 2: sending reports, which are then combined by coordinators into summaries,

Round 3: sending summaries by coordinators, updating local knowledge, and deciding on the mode of the next phase,

Round 4: a pause to synchronize phases.

Each processor, including coordinators, sends a message about the tasks done to every coordinator that either proclaimed itself a coordinator or confirmed its existence in the previous phase. If a processor is sure that all the tasks have been completed, then it halts. If no coordinator is heard during a phase then the remaining operational processors start an election processor to select new coordinators, this is done by switching to the election mode. During an election phase, a worker may decide to be a coordinator, then it proclaims this by broadcasting a suitable message to all the processors it expects to be operational. More precisely, an *election* phase consists of the following rounds:

Round 1: deciding whether to become a coordinator, Round 2: sending proclamations by coordinators,

Round 3: sending reports to the new coordinators, which combine them into summaries,

Round 4: sending summaries out; if at least one summary received then switching to work phase.

Local knowledge

Each processor $\mathcal{P}_v$ maintains three lists:

- list of outstanding tasks $\texttt{Tasks}_v$,
- list of operational processors $\texttt{Processors}_v$,
- list of coordinators $\texttt{Coordinators}_v$.

The contents of lists $\texttt{Tasks}_v$ and $\texttt{Processors}_v$ are overestimates of the true situation because if a processor fails then it takes a number of steps for this information to reach every other processor, similarly if a task is performed. At the start of computation, each processor $\mathcal{P}_v$ obtains a list of all the tasks and a list of all the processors, sorts the lists and initializes $\texttt{Tasks}_v$ to the list of all the tasks and $\texttt{Processors}_v$ to the list of all the processors. Then $\mathcal{P}_v$ sets $\texttt{Coordinators}_v$ to be empty. Message M_v that processor $\mathcal{P}_v$ sends includes the following data: the identifier v of $\mathcal{P}_v$, the coordinator/worker status of $\mathcal{P}_v$, and lists $\texttt{Tasks}_v$ and $\texttt{Processors}_v$.

Each processor $\mathcal{P}_v$ maintains the following local variables:

$\texttt{level}_v$, $\texttt{coordination\_priority}_v$, $\texttt{phase}_v$.

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

The algorithm uses two special functions. Function **random**(p) outputs a random integer value from interval $[1, p]$. Function **execute** switches control into the procedure given as an argument.

1) $\texttt{level}_v := 1$
2) $\texttt{coordination\_priority}_v :=$ **random**(p)
3) $\texttt{phase}_v :=$ `election`
4) initialize lists $\texttt{Processors}_v$ and $\texttt{Tasks}_v$

while not halted **do**
    if $\texttt{phase}_v =$ `work` **then execute** WORKPHASE
        else execute ELECTIONPHASE

Figure 3.1: Algorithm RA for processor $\mathcal{P}_v$.

The overall structure of algorithm RA is described in Figure 3.1.

After initialization of the variables and lists, all the processors repeat four-round phases in a loop until all of them terminate. In our description of rounds we always make them to consist of two steps: Step 1 is for communication and Step 2 is for local computations. Phases are either work or election, according to the phase mode. Figures 3.2 and 3.3 contain detailed descriptions of work and election phases respectively.

Now we consider correctness of algorithm RA. A phase is *productive* if it ends in a work phase. Let *local view* of processor $\mathcal{P}_v$ consists of the values of variables $\texttt{phase}_v$ and $\texttt{level}_v$, and lists $\texttt{Tasks}_v$, $\texttt{Processors}_v$ and $\texttt{Coordinators}_v$.

Lemma 3.1 *In an execution of* RA*, local views of any two operational processors are the same at the end of a productive phase.*

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

ROUND 1:

Step 1: pause

Step 2:

- perform a task: if the size of list $\texttt{Tasks}_v$ is at least as large as the size of list $\texttt{Processors}_v$ then choose task T such that the rank of T in $\texttt{Tasks}_v$ is equal to the rank of $\mathcal{P}_v$ in $\texttt{Processors}_v$ else select a random task T from $\texttt{Tasks}_v$
- remove T from $\texttt{Tasks}_v$

ROUND 2:

Step 1: send report M_v to every processor on list $\texttt{Coordinators}_v$

Step 2: if $\mathcal{P}_v$ is not a coordinator then pause in this step else do the following:

- receive all the reports
- remove any item T from list $\texttt{Tasks}_v$ that is not included in some list $\texttt{Tasks}_w$ contained in message M_w just received
- remove any item $\mathcal{P}_w$ on list $\texttt{Processors}_v$ such that no report of processor $\mathcal{P}_w$ was received

ROUND 3:

Step 1: if $\mathcal{P}_v$ is a coordinator then send the summary M_v to all the processors on list $\texttt{Processors}_v$

Step 2:

- receive all the summaries
- update lists $\texttt{Tasks}_v$ and $\texttt{Processors}_v$ similarly as in Round 2
- update list $\texttt{Coordinators}_v$: for any item $\mathcal{P}_w$ on this list, if no summary received from it in Round 2 then remove $\mathcal{P}_w$ from both lists $\texttt{Coordinators}_v$ and $\texttt{Processors}_v$
- if list $\texttt{Coordinators}_v$ is empty then switch the phase to `election`
- if list $\texttt{Tasks}_v$ is empty then halt

ROUND 4: pause for the whole round

Figure 3.2: WORKPHASE of algorithm RA for processor $\mathcal{P}_v$.

ROUND 1:

Step 1: pause

Step 2:

- increase `level`$_v$
- check if the test to qualify for a coordinator is passed

ROUND 2:

Step 1: if test passed then send proclamation M_v in to every processor on list `Processors`$_v$

Step 2:

- receive all the proclamations
- add $\mathcal{P}_w$ to list `Coordinators`$_v$ for each proclamation M_w received

ROUND 3:

Step 1: if list `Coordinators`$_v$ is nonempty then send report M_v to every coordinator on it

Step 2: if $\mathcal{P}_v$ is a new coordinator then do the following:

- receive reports
- update lists `Tasks`$_v$ and `Processors`$_v$

ROUND 4:

Step 1: if $\mathcal{P}_v$ is a coordinator then send summary M_v to all the processors on list `Processors`$_v$

Step 2:

- receive all the summaries
- if at least one summary received then update three lists `Tasks`$_v$, `Processors`$_v$ and `Coordinators`$_v$
- if list `Coordinators`$_v$ nonempty then switch to phase `work`

Figure 3.3: ELECTIONPHASE of algorithm RA for processor $\mathcal{P}_v$.

Lemma 3.2 *Algorithm* RA *terminates after at most* $t + p + 2\log p$ *phases.*

Proof: The number of election phases is at most $\log(1-c)p$ because, by Lemma 3.1, the contents of variables `level` are simultaneously increased during each such a phase. Each work phase that is not productive precedes an election phase, hence the number of such phases is not larger than the number of election phases. It is enough to count the productive phases. Observe that during such a phase each processor $\mathcal{P}_v$ either removes a task from $\mathtt{Tasks}_v$ or adds a processor to $\mathtt{Coordinators}_v$. Namely, if it is a work phase, then each processor performs a task and removes it from its list of tasks, at the end of this phase all the processors have removed the task from their lists of tasks by Lemma 3.1. If this is an election phase then new coordinators are added to $\mathtt{Coordinators}_v$. It follows that the number of such phases is at most $t + p$. □

Theorem 3.3 *Algorithm* RA *is reliable and solves Do-All.*

Proof: By Lemma 3.2, algorithm RA terminates. Processor $\mathcal{P}_v$ terminates if list $\mathtt{Tasks}_v$ is empty. Processor $\mathcal{P}_v$ has removed a task from this list either if it performed it itself or received information in a summary that it had been performed. □

3.2 Analysis of algorithm RA

Algorithm RA operates by processors selecting randomly tasks to be performed during computation. This can be modeled by the following process:

> Clearing s urns with n balls:
>
> At the start there are s empty urns. A step begins with a random placement of n balls in the given set of urns. Then the urns containing at least one ball are removed. Such steps are iterated until no urn remains.

Lemma 3.4 *The process of clearing s urns with n balls, for $n \geq s$, terminates within*

$$\log^* n - \log^* \frac{n}{s} + \mathcal{O}(1)$$

steps with the probability at least $1 - e^{-n^{1/5}}$, for sufficiently large number n.

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

$$\begin{aligned} m \cdot \left(\frac{m-1}{m}\right)^k &= m \cdot \left(1 - \frac{1}{m}\right)^{m \cdot \frac{k}{m}} \\ &\leq m \cdot e^{-k/m} \ . \end{aligned}$$

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

We consider the expectancies

$$Y_{i,k} = \mathsf{E}\left[X_{i,k} \mid X_{i,1}, \ldots, X_{i,k-1}\right] .$$

Then the sequence $\langle Y_{i,k} \rangle_k$ is a martingale (cf. [31]). Adding one ball changes the number of empty urns by at most 1, hence $|Y_{i,k+1} - Y_{i,k}| \leq 1$. This allows to use the method of bounded differences (cf. [51]), and to estimate probabilities of deviations by the Azuma inequality, which takes the form

$$\mathsf{P}\left[X_i - \mathsf{E}\left[X_i\right] \geq x\right] \leq \exp(-x^2/2n) \ , \tag{3.1}$$

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

Let $e^{(k+1)} = e^{e^{(k)}}$, where $e^{(1)} = e$. We have $\mathsf{E}\left[X_1\right] \leq n/e$. Then by (3.1) inequality

$$\mathsf{P}\left[X_1 > \frac{n}{e} + \epsilon_1 n\right] \leq e^{-\delta_1 n}$$

holds for each sufficiently small $\epsilon_1 > 0$ and some corresponding $\delta_1 > 0$. Let event $\mathcal{A}_1$ hold iff $X_1 < \frac{n}{e} + \epsilon_1 n$. Then

$$\begin{aligned} \mathsf{E}\left[X_2 \mid \mathcal{A}_1\right] &\leq n(e^{-1} + \epsilon_1) \exp\bigl(-\frac{n}{n(e^{-1} + \epsilon_1)}\bigr) \\ &\to \frac{n}{e \cdot e^{(2)}} \ , \end{aligned}$$

with $\epsilon_1 \to 0$. Let event $\mathcal{A}_2$ hold iff $X_2 < n/e^{(2)}$. Then

$$\mathsf{P}\left[-\mathcal{A}_2 \mid \mathcal{A}_1\right] < e^{-\delta_2 n} \ ,$$

for a certain $\delta_2 > 0$. The case of X_3 is similar to X_2, namely

$$\mathsf{E}\left[X_3 \mid \mathcal{A}_1 \cap \mathcal{A}_2\right] \leq n/(e^{(2)} e^{(3)}) \ .$$

Let $\mathcal{A}_3$ hold iff $X_3 < n/e^{(3)}$. We can estimate the probability similarly as before:

$$\mathsf{P}\left[-\mathcal{A}_3 \mid \mathcal{A}_1 \cap \mathcal{A}_2\right] < \exp\bigl(-n/2(e^{(3)})^4\bigr) \ .$$

This leads to a generalization: define $\mathcal{A}_k$ to hold iff $X_k < n/e^{(k)}$ and

$$a_k = \mathsf{P}\left[-\mathcal{A}_k \mid \mathcal{A}_1 \cap \cdots \cap \mathcal{A}_{k-1}\right] .$$

Then the following inequalities hold:

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

and

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

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

$$\begin{aligned}\prod_{3\le i<u}(1-a_i) &= \exp\big(\ln \prod_{3\le i<u}(1-a_i)\big)\\ &= \exp\big(\sum_{3\le i<u}\ln(1-a_i)\big)\\ &\ge \exp(-2\sum_{3\le i<u} a_i) ,\end{aligned}$$

for $a_i < 1/2$. Also

$$\begin{aligned}a_i &< \exp(-n/2(e^{(i)})^4)\\ &\le \exp(-n/2\log^4 n) ,\end{aligned}$$

for $3 \le i < u$. Hence

$$\sum_{3\le i<u} a_i < \log^* n \cdot \exp(-n/2\log^4 n) .$$

We need to estimate $\mathsf{P}[\bigcap_{1\le i<\log^* n}\mathcal{A}_i]$. This number is at least

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

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

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

$$\begin{aligned}\mathsf{E}[Z] &= \frac{n}{\log n}\cdot e^{-\log n}\\ &= \Theta(1/\log n) .\end{aligned}$$

From the Azuma inequality:

$$\mathsf{P}[Z - \mathsf{E}[Z] \ge d\sqrt{n}] \le \exp(-d^2/2) .$$

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

$$\mathcal{O}((\frac{n^{3/4}-1}{n^{3/4}})^n) = \mathcal{O}(e^{-n^{1/4}}) .$$

With probability $\mathcal{O}(n^{3/4} \cdot e^{-n^{1/4}})$ there is an empty urn after that step.

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

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

for sufficiently large n.

Finally consider the case $s < n$. We need to count the number of steps needed to have at most $n/\log n$ empty urns. During each of these steps we estimate the conditional expected number of the remaining urns by number of the form

$$\frac{s}{e^{(k-1)}e^{(k)}} ,$$

which is less than $n/\log n$ for

$$k = \log^* n - \log^* \frac{n}{s} + \mathcal{O}(1) .$$

The rest of the proof is the same as in the case $n = s$. □

The next two theorems state in quantitative terms the fact that algorithm RA is very efficient against WEAKLY-ADAPTIVE LINEARLY-BOUNDED adversaries with any constant $1 - c$, where $0 < 1 - c < 1$.

Theorem 3.5 *Algorithm* RA *solves Do-All with work*

$$\mathcal{O}(t + p \cdot (1 + \log^* p - \log^* \frac{p}{t}))$$

against any WEAKLY-ADAPTIVE LINEARLY-BOUNDED *adversary.*

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

$$\mathcal{O}(p(1 + \log^* p - \log^* \frac{p}{t}))$$

by Lemma 3.4.

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

$$
\begin{aligned}
\mathsf{E}\left[X\right] &= \sum_{i=1}^{p} \mathsf{P}\left[X \geq i\right] \\
&\leq \sum_{i=1}^{p} \Big(1 - \frac{i-1}{p}\Big)^{cp} \\
&\leq \sum_{i=1}^{p} e^{-c(i-1)} \\
&= \mathcal{O}(1) \ .
\end{aligned}
$$

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

Theorem 3.6 *The expected number of messages sent while* RA *is executed against any* WEAKLY-ADAPTIVE LINEARLY-BOUNDED *adversary is*

$$\mathcal{O}(t + p \cdot (1 + \log^* p - \log^* \frac{p}{t}) \ .$$

Proof: Let X be the minimum value of variable $\texttt{level}_v$ for all the processors $\mathcal{P}_v$ that never fail. Let Y be the number of processors $\mathcal{P}_w$ such that inequality $\texttt{level}_w \leq 2^i$ holds, where i is the integer satisfying $2^{i-1} < X \leq 2^i$. Number $\mathsf{E}\left[Y\right]$ is the expected number of coordinators in the course of algorithm. The expected number of phases is

$$\mathcal{O}(1 + \frac{t}{p} + \log^* p - \log^* \frac{p}{t}) \ .$$

The expected number of messages sent during a phase is $\mathcal{O}(p \cdot \mathsf{E}[Y])$. We estimate $\mathsf{E}[Y]$ as follows:

$$
\begin{aligned}
\mathsf{E}[Y] &= \sum_{i=1}^{p} \mathsf{E}[Y \mid X = i]\mathsf{P}[X = i] \\
&\leq \sum_{j=1}^{1+\log p} 2^j \mathsf{P}[2^{j-1} < X \leq 2^j] \\
&\leq \sum_{j=1}^{1+\log p} 2^j \left(1 - \frac{2^{j-1}}{p}\right)^{cp} \\
&\leq \sum_{j=1}^{1+\log p} 2^j \exp\left(-c2^{j-1}\right) \\
&\leq \frac{1}{2} \sum_{k=1}^{\infty} k e^{-ck} \\
&= \mathcal{O}(1) \ .
\end{aligned}
$$

Straightforward calculations complete the proof. □

3.3 Lower bound

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

The strategy S_a is described by the following two rules:

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

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

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

Lemma 3.7 *If $t \leq 2p$ then for any algorithm there is strategy S_a which forces time to be*

$$\Omega(\frac{\log t}{\log \log t}) \ .$$

Proof: Take strategy S_a for $a = \frac{1}{\log t}$. Only Rule 2 is applied. Then

$$\lceil a t_{i-1} \rceil \geq t_i \geq a t_{1-1}$$

by Rule 2. The number of tasks performed in phase i satisfies

$$t_{i-1} - t_i \leq (1-a) t_{i-1} \ ,$$

and these tasks can be selected in such a way that the average number of tasks per processor is not greater than p_{i-1}/t_{i-1}. Hence

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

It follows by induction that $t_n \geq a^n \cdot t$ and $p_n \geq (1-a)^n \cdot p$. Substitute

$$n = \frac{\log t}{\log \log t} \ .$$

Then

$$\begin{aligned} t_n &\geq (\frac{1}{\log t})^{\frac{\log t}{\log \log t}} \cdot t \\ &= 1 \end{aligned}$$

and

$$\begin{aligned} p_n &\geq (1 - \frac{1}{\log t})^{\frac{\log t}{\log \log t}} \cdot p \\ &\geq 4^{-\frac{1}{\log \log t}} \cdot p \\ &\geq cp \ , \end{aligned}$$

for sufficiently large t. □

Lemma 3.8 *If $t > 2p$ then for each algorithm there is a strategy S_a that forces work to be*

$$\Omega(t + \frac{p \log p}{\log \log p}) \ .$$

Proof: Apply strategy S_a for $a = \frac{1}{\log p}$. While Rule 1 is applied, the amount of work is at least $t - 2p$ because each task is performed by at most one processor. Consider the phase when Rule 2 starts to be applied. Then the number t' of still outstanding tasks satisfies $p < t' \leq 2p$. The problem is reduced to the case $t \leq 2p$, and Lemma 3.7 applies. $\square$

Theorem 3.9 *Any algorithm requires work*

$$\Omega(t + p \cdot \frac{\log t}{\log \log t})$$

against the adversary STRONGLY-ADAPTIVE LINEARLY-BOUNDED.

Proof: It follows from Lemmas 3.7 and 3.8 by asymptotic calculations. If $t \leq 2p$, then apply Lemma 3.7, because inequality

$$t < \frac{p \log t}{\log \log t}$$

holds. If the following inequalities hold

$$\frac{t}{\log t} \leq 2p < t \ ,$$

then use Lemma 3.8 and estimation

$$\begin{aligned} \frac{p \log p}{\log \log p} &= \Omega\Big(\frac{p \log(t/\log t)}{\log \log t}\Big) \\ &= \Omega\Big(\frac{p \log t}{\log \log t}\Big) \ . \end{aligned}$$

If $p < \frac{t}{\log t}$ then

$$\begin{aligned} \frac{p \log t}{\log \log t} &< \frac{t}{\log t} \cdot \frac{\log t}{\log \log t} \\ &< t \ . \end{aligned}$$

This combined with Lemma 3.8 gives

$$\begin{aligned} \Omega\Big(t + \frac{p \log p}{\log \log p}\Big) &= \Omega(t) \\ &= \Omega\Big(t + \frac{p \log t}{\log \log t}\Big) \end{aligned}$$

of work. $\square$

Note that this result can be extended to the UNBOUNDED adversary by using the same strategy.

1) $\texttt{blocksize}_v := 0$
2) $\texttt{Block}_v := \emptyset$
3) initialize lists $\texttt{Processors}_v$ and $\texttt{Tasks}_v$.

while not halted **do execute** ModWorkPhase

Figure 3.4: Algorithm DA for processor $\mathcal{P}_v$.

3.4 Deterministic algorithm

In this section we develop and analyze algorithm DA whose work performance matches the lower bound for work given in Theorem 3.9. The algorithm is similar to RA, the main differences are as follows:

- all the processors act as coordinators;
- there are no election phases in DA, only work phases performed in a loop; work phase consists of the first three rounds of the work phase of RA, with $\texttt{Processors}_v$ playing the role of $\texttt{Coordinators}_v$, and the variables needed for elections not used;
- tasks are performed in a load-balanced way, that is, processor $\mathcal{P}_v$ of rank i in list $\texttt{Processors}_v$ performs the block of $\lceil|\texttt{Processors}_v|/|\texttt{Tasks}_v|\rceil$ tasks of the ith rank, modulo the length of $\texttt{Tasks}_v$, while the last block may possibly be shorter.

Algorithm DA is given in a Figure 3.4, it refers to a phase description which is presented in detail in Figure 3.5.

The correctness of DA is proved similarly as the correctness of RA. There are two differences: we do not consider election phases and the notion of a productive phase should be redefined by adding 'and $\texttt{blocksize}_v \leq 0$'.

Now we count the amount of work performed by algorithm DA against any Linearly-Bounded adversary. Let p_i be the number of processors still operational after productive phase i, where $p_0 = p$. Let t_i be the number of outstanding tasks after productive phase i, where $t_0 = t$. Consider a sequence $\alpha = \langle a_i \rangle$, where $i = 1, \ldots, n$ and $0 < a_i \leq 1$, for each $1 \leq i \leq n$. A pair of sequences $\langle p_i \rangle, \langle t_i \rangle$, where $i = 0, \ldots, n$,

ROUND 1:

Step 1: pause

Step 2:

- if $\texttt{blocksize}_v \leq 0$ then set $\texttt{blocksize}_v$ to the value of $\lceil|\texttt{Processors}_v|/|\texttt{Tasks}_v|\rceil$
- partition $\texttt{Tasks}_v$ into consecutive blocks of size $\texttt{blocksize}_v$ except possibly for the last one
- set variable $\texttt{Block}_v$ to the block which rank is the same as the rank of $\mathcal{P}_v$ in $\texttt{Processors}_v$
- if $\texttt{Block}_v \neq \emptyset$ then perform the first task from $\texttt{Block}_v$ and remove it from lists $\texttt{Block}_v$ and $\texttt{Tasks}_v$
- decrease $\texttt{blocksize}_v$ by 1

ROUND 2:

Step 1: if $\texttt{blocksize}_v \leq 0$ then send report M_v to every process

Step 2:

- receive all the reports
- remove any item T from list $\texttt{Tasks}_v$ such that it is not included in some list $\texttt{Tasks}_w$ contained in message M_w just received
- remove any item $\mathcal{P}_w$ on list $\texttt{Processors}_v$ from which no report was received

ROUND 3:

Step 1: if $\texttt{blocksize}_v \leq 0$ then send the summary M_v to all the processors on list $\texttt{Processors}_v$

Step 2:

- receive all the summaries
- update lists $\texttt{Tasks}_v$ and $\texttt{Processors}_v$ similarly as in Round 2
- if list $\texttt{Tasks}_v$ is empty then halt

Figure 3.5: Phase MODWORKPHASE of algorithm DA for processor $\mathcal{P}_v$.

is an *α-scenario of length n* if the following conditions hold:

$$\begin{aligned}
&t_0 = t \ , \\
&p_0 = p \ , \\
&t_n \geq 1, \\
&p_i \leq \lceil \frac{p_{i-1}}{t_{i-1}} \rceil t_{i-1}(1 - a_i)
\end{aligned}$$

and

$$\lceil t_{i-1} \cdot a_i \rceil \geq t_i \geq t_{i-1} \cdot a_i \ ,$$

for $i = 1, \ldots, n$, and $p_i \geq cp$ for $i = 0, \ldots, n$. A *scenario* is an α-scenario for a certain suitable α.

Lemma 3.10 *Let $0 < a_1, \ldots, a_n < 1$. If $a^n = \prod_{i=1}^n a_i$ then*

$$(1 - a)^n \geq \prod_{i=1}^{n} (1 - a_i) \ .$$

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

$$n \cdot \ln(1 - a) \geq \sum_{i=1}^{n} \ln(1 - a_i) \ . \tag{3.2}$$

The left-hand side of (3.2) is

$$-n(a + \frac{a^2}{2} + \frac{a^3}{3} + \ldots) \ .$$

The right-hand side of (3.2) is

$$-(\sum_{i=1}^{n} a_i + \frac{1}{2} \sum_{i=1}^{n} a_i^2 + \frac{1}{3} \sum_{i=1}^{n} a_i^3 + \ldots) \ .$$

The inequalities

$$a^k \leq \frac{1}{n} \sum_{i=1}^{n} a_i^k$$

hold for all integer $k > 0$ because a^k is the geometric mean of $a_1^k, \ldots, a_n^k$, while $\frac{1}{n} \sum_{i=1}^{n} a_i^k$ is their arithmetic mean. Compare the expansions of both sides of (3.2) to obtain the inequality. □

Lemma 3.11 *Let $\langle p_i \rangle, \langle t_i \rangle$ be an α-scenario of length n. Then the following inequalities hold, for $1 \leq i \leq n$:*

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

and

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

Proof: We estimate p_i as follows:

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

and by induction obtain

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

Numbers t_i satisfy the following inequalities:

$$\begin{aligned} t_i &\leq t_{i-1} \cdot a_i + 1 \\ &\leq [t_{i-2} \cdot a_{i-1} + 1] a_i + 1 \\ &\leq t_{i-2} \cdot a_{i-1} \cdot a_i + 2 \ , \end{aligned}$$

we obtain

$$\begin{aligned} t \prod_{j=1}^{i} a_j &\leq t_i \\ &\leq t \prod_{j=1}^{i} a_j + i \end{aligned}$$

by induction. □

Lemma 3.12 *If $t \leq cp$ then any α-scenario of length n satisfies*

$$n \leq \frac{2 \log t}{\log \log t} \ .$$

Proof: Assume that, on the contrary,

$$n = \frac{b \log t}{\log \log t} \ ,$$

for a certain $b > 2$. By Lemma 3.10 we may take a such that

$$a^n = \prod_{i=1}^{n} a_i \ .$$

Then we have also

$$(1-a)^n \geq \prod_{i=1}^{n} (1-a_i) \ ,$$

and by Lemma 3.11 obtain

$$\begin{aligned} p_n &\leq p \prod_{i=1}^{n} (1-a_i) + t(1-a_n) \\ &\leq p(1-a)^n + t(1-a_n) \ . \end{aligned}$$

As number a we may take $h(\log t)^{-1/b}$, for a certain $h \geq 1$, because

$$\frac{1}{t} = (\frac{1}{\log^{1/b} t})^{\frac{b \log t}{\log \log t}} \ .$$

We obtain $1 \leq t_n \leq i + 1$, because

$$\begin{aligned} t \prod_{i=1}^{n} a_i + i &\geq t_n \\ &\geq 1 \end{aligned}$$

by Lemma 3.11. Then

$$\begin{aligned} (1-a)^n &= \Big(1 - \frac{h}{\log^{1/b} t}\Big)^{\frac{b \log t}{\log \log t}} \\ &\leq \exp\Big(\frac{-hb \log^{1-1/b} t}{\log \log t}\Big) \\ &\to 0 \ , \end{aligned}$$

for sufficiently large t. Finally

$$\begin{aligned} p_n &\leq p(1-a)^n + t(1-a_n) \\ &\leq p[(1-a)^n + c(1-a_n)] \\ &< cp \ , \end{aligned}$$

what contradicts the assumption $p_n \geq cp$. $\square$

Theorem 3.13 *Algorithm* DA *performs work*

$$\mathcal{O}(t + \frac{p \log t}{\log \log t})$$

in the worst case against any linearly-bounded adversary.

Proof: Let p_i and t_i be the numbers of operational processors and outstanding tasks after productive phase i when algorithm DA executed. Then $\langle p_i \rangle, \langle t_i \rangle$ is a scenario. If $t \leq cp$ then by Lemma 3.12 the work is

$$\mathcal{O}(\frac{p \log t}{\log \log t}) = \mathcal{O}(t + \frac{p \log t}{\log \log t}) \ .$$

Suppose that $t > cp$. Algorithm DA assigns at most $\lceil 1/c \rceil$ processors to each task during such a phase when there are more than cp outstanding tasks. The processors that never fail perform $\Omega(p)$ tasks during such a phase. Eventually the number of outstanding tasks becomes at most cp, while the work performed so far is $\mathcal{O}(t)$. By Lemma 3.12 the remaining work is

$$\mathcal{O}(\frac{p \log t}{\log \log t}) \ ,$$

because the function $\log x / \log \log x$ is increasing. □

Theorem 3.14 *The number of messages of algorithm* DA *is*

$$\mathcal{O}(p^2 \cdot \frac{\log t}{\log \log t}) \ .$$

Proof: There are $\mathcal{O}(1)$ productive phases after which the number of still outstanding tasks is greater than cp. The communication increases by $\mathcal{O}(p^2)$ during each productive step.

The total communication during phases with the number of unperformed tasks being at most cp is $\mathcal{O}(p^2 \log t / \log \log t)$. Note that during one phase the communication is bounded by $\mathcal{O}(p^2)$. The proof is complete by Lemma 3.12. □

Algorithm DA solves the *Do-All* problem with the same asymptotic bounds on work and communication. We can apply our analysis to failure patterns generated by the adversary Unbounded. Let us partition conceptually such a failure pattern into phases. During the lth phase, $l = 0, \cdots, \lceil \log p \rceil - 1$, we assume that the number of non-faulty processors is between $p/2^{l-1}$ and $p/2^l$. The sum of fractions $\log t / \log \log t$ over l can be estimated by a geometric series which gives the same asymptotic bound as in the case of Linearly-Bounded adversaries.

Chapter 4

Strongly adaptive adversaries

In this chapter consider the problem *Do-All* in the following setting:

> Model of communication: message-passing
>
> Multicast: not assumed to be atomic
>
> Adversaries: strongly-adaptive linearly-bounded

We emphasize 'strongly-adaptive' in the title because the main contribution of this chapter are algorithms efficient against STRONGLY-ADAPTIVE LINEARLY-BOUNDED adversary, which are complementary to the results of Chapter 3.

We start with a generic scheme of algorithms solving the problem *Do-All*, which is given in Section 4.1. It depends on an underlying communication graph and on rules used to assign tasks to processors. We give three different rules in the following sections, which yield three corresponding algorithms. All algorithms solve *Do-All* against the UNBOUNDED adversary, however their performance is optimized against the LINEARLY-BOUNDED adversary. A communication graph is the same in all the presented algorithms, the performance is determined by selection rules. The algorithms balance communication and work: they are asymptotically equal against LINEARLY-BOUNDED adversaries.

First we develop a deterministic constructive algorithm BAL in Section 4.2, then trade either determinism or constructiveness for better work and communication performance. A deterministic algorithm is said to be *constructive* if it is developed in a way giving all the implementation details, this is in contrast with a situation when an algorithm refers to certain combinatorial objects that are known to exist, but the proof of existence is not constructive, for instance if it is carried out by the probabilistic method. Algorithm BAL uses simple load-balancing to solve the problem *Do-All* attaining work and communication $\mathcal{O}(t + p \cdot \sqrt{t+p} \cdot \log^{3/2} p)$ against the LINEARLY-

Bounded adversary. Randomized algorithm RAL described in Section 4.3 solves *Do-All* with work and communication $\mathcal{O}(t+p\cdot\log^2 p/\log\log p)$ against the Strongly-Adaptive Linearly-Bounded adversary.

In Section 4.4 we present an efficient but not constructive deterministic algorithm DPAL which solves the problem *Do-All* with work and communication $\mathcal{O}(t+p\cdot\log^2 p)$ against the Linearly-Bounded adversary. The performance of this more efficient algorithm among deterministic ones is close to the lower bound $\Omega(t+p\log t/\log\log t)$ on work proved in Chapter 3. The algorithm uses a family of permutations determining the orders in which processors perform tasks. The existence of such families is proved by the probabilistic method.

4.1 Overview of algorithms

4.1.1 Communication and graphs

Processors may choose to send messages to only a few other processors in a communication step in order to save on communication. Each processor will use a fixed set of links for this purpose, which are the edges of (a sparse) *communication graph* $G = G(p) = (V, E)$. In this graph the set of all the processors is the set V of its nodes, and the respective subset of links is the set E of edges. Actually $G(p)$ is the initial communication graph, each faulty or halted processor is immediately removed from it, so the communication graph evolves through a sequence of subgraphs until it eventually disappears altogether. In the course of computations the processors may have different views on the current communication graph, depending on their local knowledge. Additionally, in some communication steps a processor may send special messages to certain ad-hoc vertices which may not be its neighbors in the communication graph. This is possible because the underlying communication network is a complete graph and the only purpose to use sparse communication graphs is to save on the total number of point-to-point messages.

To specify graphs $G(p)$ we will refer to a number of parameters. The constant c is the parameter used in the definition of a Linearly-Bounded adversary, we require that $0 < c < 1$. Then there are ϱ and γ, which depend on c, and we require that $0 < \varrho < c < 1$ and $\gamma > 0$. We use the notation $g(x)$ to abbreviate $\gamma \lg x$, where $\lg x$ is the logarithm to the base 2.

First we describe useful properties and fix terminology concerning graphs. Let $G = (V, E)$ be a graph. A *subgraph* always denotes an induced subgraph. If $H \subseteq V$ is a subset of nodes of G then the subgraph induced by H is also denoted by the notation H. Sometimes by H we mean the graph induced by H in a subgraph G' of graph G, then this G' is mentioned explicitly. The number of nodes in subgraph H is denoted by $|H|$. The notation $N_G(H)$ means the subset of V consisting of all the nodes from $H \subseteq V$ and their neighbors in graph G. By G^k we mean the k-th power of graph G: this is the graph having V as its set of nodes, and two nodes are connected by an edge in G^k iff there is a path between them in G of length at most k. A subgraph $H \subseteq G$ is *good in* G if the size of H is at least $\varrho \cdot |V|$ and the diameter of H is at most $g(|V|) = \gamma \lg |V|$. If G' is a subgraph of graph G and $v' \in G'$ then by the *distance component of node* v *in* G' we mean the subgraph containing each node in G' whose distance from v in G' is at most $g(|V|) = \gamma \lg |V|$. A node $v \in G'$ is said to be *good in* G' if the size of its distance component in G' is at least $\varrho \cdot |V|$.

We define a family of graphs $G(n)$, each $G(n)$ of n nodes, with certain good properties. These properties depend on whether the graphs are used by deterministic or randomized algorithms. A property that we will need for deterministic algorithms is somewhat weaker than that needed for randomized ones, and is as follows:

> PROPERTY $\mathcal{D}$: For every subgraph $G' \subseteq G(n)$ of size at least $c \cdot n$, there is a good subgraph $H \subseteq G'$ in $G(n)$.

Note that all the nodes from such H are good in G'. A stronger property, that we will need for a randomized algorithm, is as follows:

> PROPERTY $\mathcal{R}$: There is a function P, which assigns subgraph $P(R) \subseteq G(n)$ to each subgraph $R \subseteq G(n)$ of size at least $c \cdot n$, such that the following hold:
>
> a) $P(R) \subseteq R$
>
> b) $|P(R)| \geq \varrho n$
>
> c) the diameter of $P(R)$ is at most $g(n) = \gamma \lg n$
>
> d) if $R_1 \subseteq R_2$ then $P(R_1) \subseteq P(R_2)$

Note that PROPERTY $\mathcal{R}$ implies PROPERTY $\mathcal{D}$. Suppose $\langle G_i \rangle$ is a sequence of subgraphs of $G(n)$ such that both $G_{i+1} \subseteq G_i$ and $|G_i| \geq c \cdot n$, for $i \geq 1$. We may use PROPERTY $\mathcal{R}$ to obtain a sequence of good subgraphs $\langle H_i \rangle$ in $G(n)$, such that $H_{i+1} \subseteq H_i$ and $|H_i| \geq \varrho n$. To this end it is sufficient to define $H_i = P(G_i) \subseteq G_i$. The specific families of graphs $G(n)$ that we will use have a uniform upper bound on the degrees of nodes, the notation Δ means the maximum node degree of each graph $G(n)$.

Construction of a family of good communication graphs

In this part we show that the family $L^k(n)$ of regular expander graphs introduced independently by Lubotzky, Phillips, Sarnak [44] and Margulis [47] satisfies Property $\mathcal{D}$

and Property $\mathcal{R}$, for some positive constant integer k dependent from parameter c. These graphs are defined and can be constructed for each number n of the form $q(q^2-1)/2$, where q is a prime integer congruent to 1 modulo 4. The node degree Δ can be any number such that $\Delta - 1$ is a prime congruent to 1 modulo 4 and a quadratic nonresidue modulo q. It follows, from the properties of the distribution of prime numbers (see e.g. [19]), that Δ can be selected to be a constant independent from number p, and number q such that $n = q(q^2-1)/2 = \Theta(p)$. Starting from now on the notation $G(p)$ will mean the suitable graph $L(n)$ for $n = \Theta(p)$, with each processor simulating $\mathcal{O}(1)$ nodes.

Upfal [57] showed that graphs $L(n)$ have the following property: if Z is a subgraph of graph $L(n)$ of size at most $n/72$ then there is an induced subgraph $H \subseteq G \setminus Z$ of size greater than $11n/12$ and diameter at most $30 \log_2 n$. It follows that we can take $\gamma = 30$ and $\varrho = 11/12$ for $c \geq 71/72$ to have Property $\mathcal{D}$. Similarly, it follows from the results from [57] that graphs $L(n)$ satisfy Property $\mathcal{R}$, for the same set of values of c, γ and ϱ, and a certain set function P'. Namely, we may take function P' defined by Upfal in [57] as follows. Let R be a subset of nodes. First define sequence $\langle Z_j \rangle$ by induction:

$Z_1 = L(n) \setminus R$, and
$Z_{j+1} = Z_j \cup \{v \in L(n) \setminus Z_j : v \text{ is a neighbor of at least } \Delta/5 \text{ nodes from } Z_j\}$.

Let k be the first number such that $Z_k = Z_{k+1}$ holds. Note that $Z_j = Z_{j+1}$ for $j \geq k$. Define $P'(R) = L(n) \setminus Z_k$. A proof that P' satisfies (a.) and (d.) in Property $\mathcal{R}$ is by induction on j. The remaining parts (b.) and (c.) follow from the proofs of Lemma 1 and Lemma 2 in [57], respectively. The constants obtained by Upfal [57] for the graphs $L(n)$ can be improved (cf. [3]). Moreover, if constant c is a small number then a suitable power $L(n)^k$, for sufficiently large k, can be taken as the communication graph. This approach was first used by Diks and Pelc [23] in the context of broadcasting and fault diagnosis. They showed that property similar to our Property $\mathcal{D}$ holds for such graphs. They proved the following technical fact about the expansion of powers of graphs $L(n)$:

Fact 4.1 ([23]) *For every constant $0 < c < 1$ there exist a positive integer constant k such that, for any subgraph R of $L(n)^k$ of size at least $c \cdot n$, inequality*

$$|N_{L(n)^k}(R)| \geq \frac{71}{72} \cdot n$$

holds. □

Graphs $L(n)^a$, for a constant a, are still constructive and of constant degree. We show, by extending the results in [57] and using Fact 4.1, that suitably large powers of graphs $L(n)$ have PROPERTY $\mathcal{R}$.

Lemma 4.2 *For every $0 < c < 1$ there exists a positive integer constant ℓ such that graph $L(n)^\ell$ has* PROPERTY $\mathcal{R}$, *with certain parameters $\varrho > 0$, $\gamma > 0$, and a function P.*

Proof: We already know that if parameter c is sufficiently large then graph $L(n)$ satisfies PROPERTY $\mathcal{R}$ with function P' and constants ϱ' and γ'. We need to determine parameters γ and ϱ for an arbitrary given $0 < c < 1$, possibly small. Let us set $\gamma = 2 + \gamma'$. Let k be from Fact 4.1. Define $\ell = 2k + 1$. Next we construct function P for graph $L(n)^\ell$. Let R be a subgraph of graph $L(n)^\ell$ of size at least $c \cdot n$. It follows from Fact 4.1 that inequality

$$|N_{L(n)^k}(R)| \geq 71n/72$$

holds. Hence we have

$$|P'(N_{L(n)^k}(R))| \geq \varrho' n \ .$$

Let A denote the subgraph

$$P'(N_{L(n)^k}(R)) \cap R$$

and B the subgraph

$$P'(N_{L(n)^k}(R)) \setminus R \ .$$

Define $P(R)$ to be the subgraph

$$P(R) = N_{L(n)^k}(A \cup B) \cap R \ .$$

One can see that property (a.) holds by the definitions of $P(R)$.

To prove (b.) we estimate $|P(R)|$. This number is at least $\varrho' n/2\Delta^k$. To see this notice first that inequality

$$|P'(N_{L(n)^k}(R))| = |A \cup B| \geq \varrho' n$$

holds. Next, the size of subgraph $N_{L(n)^k}(B) \cap R$ is at least $|B|/\Delta^k$. This is because graph $L(n)^k$ has degree at most Δ^k and moreover each node in B has at least one neighbor in R. Hence

$$\begin{aligned} |P(R)| &\geq |A \cup (N_{L(n)^k}(B) \cap R)| \\ &\geq \max\{\varrho' n/2, (\varrho' n/2)/\Delta^k\} \\ &\geq \varrho' n/2\Delta^k \ . \end{aligned}$$

Next we prove (c.). Let v_1 and v_2 be two vertices in subgraph $P(R)$ of $L(n)^\ell$. Consider two vertices v_1 and v_2 from $P(R)$. Suppose that they are from $P'(N_{L(n)^k}(R))$. Hence there is a path between node v_1 and node v_2 in subgraph $P'(N_{L(n)^k}(R))$ of $L(n)$ of length at most $\gamma' \log n$, what follows from the proof in [57]. Let

$$v_1 = w_1, w_2, \ldots, w_m = v_2$$

be a sequence of consecutive nodes on this path. We claim that there is a path between node v_1 and node v_2 of the same length in subgraph $P(R)$ of $L(n)^\ell$. We denote the consecutive nodes on this path by

$$v_1 = w'_1, w'_2, \ldots, w'_m = v_2 \ .$$

To construct this required path, we traverse the initial path

$$v_1 = w_1, w_2, \ldots, w_m = v_2$$

in $L(n)$ edge by edge and determine the consecutive nodes in subgraph $P(R)$ of $L(n)^\ell$. The construction is by induction on the length of the preceding segment of the path. Suppose that we have constructed an initial segment of this path up to node w'_i, where $1 \le i < m-1$. Next we find $w'_{i+1} \in P(R)$ such that w'_{i+1} is a neighbor of w'_i in $L(n)^\ell$, and either $w'_{i+1} = w_{i+1}$ or w'_{i+1} is a neighbor of w_{i+1} in $L(n)^k$. There are four cases to consider:

a) Suppose $w_i = w'_i$ and w_{i+1} is in graph $P(R)$. Let $w'_{i+1} = w_{i+1}$. Nodes w'_i and w'_{i+1} are neighbors in $L(n)$ and so also in $L(n)^\ell$.

b) Suppose $w_i = w'_i$ and w_{i+1} is not in graph $P(R)$. Then $w_{i+1} \in B$ and let w'_{i+1} be any neighbor of w_{i+1} which is in subgraph

$$(N_{L(n)^\ell}(B) \setminus B) \cap R \subseteq P(R)$$

of $L(n)^k$. Nodes w'_i and w'_{i+1} are neighbors in $L(n)^\ell$, because their distance in $L(n)$ is $1 + k < \ell$.

c) Suppose $w_i \neq w'_i$ and w_{i+1} is in graph $P(R)$. Hence, by the inductive assumption, node w_i is a neighbor of w'_i in graph $L(n)^k$. We can take w_{i+1} as w'_{i+1}. The distance in graph $L(n)$ between w'_i and w'_{i+1} is $k+1 < \ell$, hence there is an edge between them in graph $L(n)^\ell$.

d) Suppose $w_i \neq w'_i$ and w_{i+1} is not in graph $P(R)$. Hence, by the inductive assumption, node w_i is a neighbor of w'_i in graph $L(n)^k$. Then $w_{i+1} \in B$ and let w'_{i+1} be any neighbor of w_{i+1} in subgraph

$$(N_{L(n)^\ell}(B) \setminus B) \cap R \subseteq P(R)$$

of $L(n)^k$. The distance in graph $L(n)$ between w'_i and w'_{i+1} is $k + 1 + k = \ell$, hence there is an edge between them in graph $L(n)^\ell$.

Note that if path $v_1 = w'_1, w'_2, \ldots, w'_{m-1}$ has been built, then the node w'_{m-1} is in set $N_{L(n)^k}(\{w_{m-1}\})$ by the inductive assumption. Hence the distance in graph $L(n)$ between the nodes w'_{m-1} and v_2 is at most $k+1$. This completes the inductive construction. The case when either v_1 or v_2 are not in $P'(N_{L(n)^k}(R))$ is straightforward, as both of them have neighbors in $P'(N_{L(n)^k}(R))$ in subgraph $L(n)^k$, by the inclusion

$$P(R) \subseteq N_{L(n)^k}(P'(N_{L(n)^k}(R))) \ .$$

It follows that the distance between them in the subgraph $P(R)$ of $L(n)^\ell$ is at most $2 + \gamma' \log n$.

To prove (d.) let us consider $R_1 \subseteq R_2$, both subgraphs of sizes at least $c \cdot n$. For any subsets of nodes A and B, let A_1 and B_1 denote the subgraphs induced by A and B, respectively, in subgraph R_1. Similarly, let A_2 and B_2 denote the subgraphs induced by A and B, respectively, in subgraph R_2. It is clear that

$$N_{L(n)^k}(R_1) \subseteq N_{L(n)^k}(R_2) \ .$$

The inclusion

$$P'(N_{L(n)^k}(R_1)) \subseteq P'(N_{L(n)^k}(R_2))$$

holds because graph $L(n)$ satisfies Property $\mathcal{R}$ with function P', for sufficiently large parameter c. Then $A_1 \subseteq A_2$ and $B_1 \subseteq B_2 \cup (A_2 \setminus A_1)$ hold too. We obtain the inclusion $P(R_1) \subseteq P(R_2)$, which completes the proof. □

4.1.2 Scheme of algorithms

In this section we present a generic scheme giving algorithms which are correct against the Unbounded adversary and can be efficient against the Strongly-Adaptive Linearly-Bounded adversary. Algorithms produced in this way are called *selection algorithms*. We assume that the parameter c is fixed. The corresponding graph G and parameters ϱ and γ are parts of the specification of the algorithms.

We call a processor *busy* at a certain round if its local knowledge does not imply that all the tasks have already been done. Selection algorithms have the property that if some tasks are still outstanding then all the operational processors are both active (have not halted) and busy.

Local knowledge

Each processor $\mathcal{P}_v$ maintains three ordered lists: list $\mathtt{Tasks}_v$ of tasks assumed to be still outstanding, list $\mathtt{Processors}_v$ of processors assumed to be active, and list $\mathtt{Busy}_v$ of processors assumed to be still busy performing tasks. These lists are initially ordered by the identifiers of the items. The position of an item in a list is its *rank* in the list. Items may be removed from lists, which affects the ranks of the remaining items. Each item on list $\texttt{Processors}_v$ has attached a *distance variable*, which stores an estimate of the distance from $\mathcal{P}_v$ to this item in the communication graph.

The information stored on these lists is not necessarily up to date in the current step. It may be an overestimate in the cases of lists $\texttt{Processors}_v$ and $\texttt{Busy}_v$, when some of the processors on these lists are either not active or not busy, respectively, but process $\mathcal{P}_v$ has not learned about it yet. Similarly, it may be an overestimate in the case of list $\texttt{Tasks}_v$, since some of the tasks have already been performed but process $\mathcal{P}_v$ still needs to get to know about it.

Each processor $\mathcal{P}_v$ maintains a counter (of phases) in its variable $\texttt{Counter}_v$. Additionally, processor $\mathcal{P}_v$ maintains local variable $\texttt{done}_v$, which remains false until $\mathcal{P}_v$ learns that all the tasks have been performed. Variable $\texttt{Selected\_Processor}_v$ is used to remember a processor. Variable $\texttt{phase}_v$ has the value `Main` or `Closing`, depending on the kind of the current phase performed by $\mathcal{P}_v$.

Messages

A message sent by processor $\mathcal{P}_v$ includes the identification v of $\mathcal{P}_v$, and (part of) its local knowledge: lists $\texttt{Tasks}_v$, $\texttt{Processors}_v$, and $\texttt{Busy}_v$. Additionally, the signal $\texttt{stop}_v$ is sent if needed.

Selection from a list

If, in the course of an algorithm, an item from a list needs to be selected, then this is performed by applying some generic `Selection_Rule`. We will use a number of rules, they will be specified later on.

Updating lists

Each active processor $\mathcal{P}_v$ updates the lists $\texttt{Tasks}_v$, $\texttt{Processors}_v$ and $\texttt{Busy}_v$ after receiving messages from its neighbors. If an item is not on the copy of a list received from some neighbor then it is removed immediately from the corresponding local list. If no message is received from a neighbor $\mathcal{P}_u$ then UID u is removed from the list $\texttt{Processors}_v$. If signal $\texttt{stop}_u$ is received from process $\mathcal{P}_u$ then ID u is removed from the list $\texttt{Busy}_v$.

Distances among processors

Processors estimate distances among them in the communication graph. The distances may change in the course of an algorithm, due to failures. During each phase we update the distance variables attached to the items in the lists $\texttt{Processors}_v$: for each processor we take the minimum of the distance variables attached to this

```
1) Counter:= t
2) done:=false
3) phase:=Main
4) initialize lists Processors_v, Busy_v and Tasks_v

while not halted do
    if phase_v = Main then execute MainPhase
                      else execute ClosingPhase
```

Figure 4.1: Generic selection algorithm for processor $\mathcal{P}_v$.

processor on all the received processors lists, then we increment it by one.

Counting the number of processors in the local distance component

During each phase a processor checks the size of the distance component it belongs to, we call it the *own distance component*. This number is estimated using the updated $\texttt{Processors}_v$ list, by comparing the distance variables with $g(p)$: if the value of the distance variable is not greater than $g(p)$ then the process is included in the component and one added to the estimate. This estimate is actually an underestimate because of possible failures and halts, which may increase the distances. If the estimated size is at least $\varrho \cdot p$ then this processor is considered to be good.

Generic selection algorithm

The algorithm is a loop which iterates phases. Each phase consists of one round. Phases are of two kinds: Main Phase and Closing Phase. The whole scheme is described in Figure 4.1.

The algorithm starts with each processor initializing its lists: $\texttt{Tasks}_v$ to a list of all the tasks, $\texttt{Busy}_v$ to a list of all the processors, except for itself, and $\texttt{Processors}_v$ to a list of all the processors. Distance variables are initialized to the true distances in the communication graph G. Finally, variables $\texttt{Counter}_v$ is initialized to t, $\texttt{phase}_v$ to `Main` and $\texttt{done}_v$ to `false`. After switching to Closing Phase, list $\texttt{Tasks}_v$ is needed no more, so it is maintained as an empty list.

During the Main Phase processor $\mathcal{P}_v$ performs task which is still outstanding according to its local knowledge. If some processor acquires the knowledge that all the tasks have been done, then it switches to Closing Phase and its new tasks are now to let other processors know this. To this end it sends `stop` signal to selected processors. After receiving such a `stop` signal a processor needs to decide if to halt or

Step 1: multicast messages to these neighbors in the communication graph that are still in the list $\texttt{Processors}_v$

Step 2: receive messages and do the following:

- update the lists $\texttt{Tasks}_v$, $\texttt{Processors}_v$ and $\texttt{Busy}_v$
- update the distance variables, attached to the processors in the list $\texttt{Processors}_v$, and estimate the size of the own distance component
- if some processor is in list $\texttt{Processors}_v$ and not in list $\texttt{Busy}_v$ then set variable $\texttt{done}_v$ to true
- if list $\texttt{Tasks}_v$ is nonempty then select a task from it by applying `Selection_Rule`, remove the task from list $\texttt{Tasks}_v$, perform the task, and decrease $\texttt{Counter}_v$ by 1
- if list $\texttt{Tasks}_v$ is empty then set `done` to true and switch `phase` to `Closing`
- if either (`done` is true and v is not considered to be good) or ($\texttt{Counter} \leq 0$) then halt

Figure 4.2: Generic MainPhase of a selection algorithm for processor $\mathcal{P}_v$.

not. The decision to continue work is made only when the processor is a good node in the communication graph and there is still some work to do according to its local knowledge. Phases are given in Figure 4.2 and Figure 4.3.

Proof of correctness

We show that a selection algorithm is correct against the adversary Unbounded, if any generic selection rule is applied. To this end we prove that every active processor will stop eventually, either by halting or by failing, and if any processor has decided to halt then all the tasks have been done.

Lemma 4.3 *A selection algorithm is reliable and solves Do-All.*

Proof: We proceed be a sequence of statements:

1. If a processor receives information that a task has been performed then this is true.

This follows by induction on the number of phases. A processor which performs

Step 1:

- multicast messages to these neighbors in the communication graph that are still in the list $\texttt{Processors}_v$
- send the signal `stop` to $\texttt{Selected\_Processor}_v$

Step 2: receive messages and do the following:

- update the lists $\texttt{Processors}_v$ and $\texttt{Busy}_v$.
- update the distance variables, attached to the processors in the list $\texttt{Processors}_v$, and estimate the size of the own distance component
- if list $\texttt{Busy}_v$ is nonempty then select a processor from it by applying `Selection_Rule`, set $\texttt{Selected\_Processor}_v$ to this processor, remove this processor from list $\texttt{Busy}_v$, and decrease $\texttt{Counter}_v$ by 1
- if either (list $\texttt{Busy}_v$ is empty) or (v is not considered to be good) or ($\texttt{Counter}_v \leq 0$) then halt

Figure 4.3: Generic ClosingPhase of a selection algorithm for processor $\mathcal{P}_v$.

a selected task is the one who deletes it from the local task list. Then this information proliferates.

2. The algorithm performs all the tasks even if one processor survives.

To show this, consider processor $\mathcal{P}_v$ when it halted. Then either $\texttt{done}_v$ is true or $\texttt{Counter}_v \leq 0$ holds. In any case $\texttt{Tasks}_v$ is empty.

3. Every processor halts eventually.

Every operational processor maintains a counter of phases. The counter is decreased in each phase. The processor halts at the latest when the counter value is equal to zero.

This completes the proof. □

The `Selection_Rule` to select elements from the lists is mean to be generic. We will consider various selection rules to perform tasks, which yield specific selection algorithms.

4.1.3 Introduction to analysis

Performance of a selection algorithm is measured against the adversary Strongly-Adaptive Linearly-Bounded. The total communication is $\mathcal{M} = \mathcal{O}(\Delta \cdot \mathcal{W})$, because during each round of a processor contributing a unit to work $\mathcal{W}$ there are at most $\Delta + 1$ messages sent, and since $\Delta = \mathcal{O}(1)$ we obtain that the communication is $\mathcal{O}(\mathcal{W})$.

For the purpose of the analysis, the computation is partitioned into epochs: one *epoch* $\mathcal{E}_i$ is defined to comprise $g(p)$ phases. The first epoch $\mathcal{E}_1$ starts in the first phase, the second epoch $\mathcal{E}_2$ in the $(g(p)+1)$-th phase and so on. One epoch is sufficient for all the processors in a good subgraph to communicate.

The whole graph G cannot always be used in a computation, because some of the nodes may fail and the other may halt and so do not participate in communication. Throughout the rest of the paper the notation G_i means the subgraph of graph G induced by the vertices that are active during the first phase of epoch $\mathcal{E}_i$. Note that $G_{i+1} \subseteq G_i$ for $i \geq 1$. Similarly, the notation H_i means a good subgraph of G_i in G, such that $H_{i+1} \subseteq H_i$ for $i \geq 1$.

In every epoch $\mathcal{E}_i$ there is a good node in G_i. This fact, in terms of communication, means that in every round there are at least $\varrho \cdot p$ operational processors which can communicate with each other during the next $g(p)$ phases, provided all of them are still active. If a processor $\mathcal{P}_v$ halts then its list $\texttt{Busy}_v$ is empty and this information needs $g(n)$ phases to proliferate. It follows that if failures are controlled by the adversary STRONGLY-ADAPTIVE LINEARLY-BOUNDED with parameter c then there are at least $\varrho \cdot p$ processors, which stay active till the end of the computation, possibly with the exception of the last $g(p)$ rounds, and can communicate with each other with the delay of at most $g(p)$ phases.

Let $T_{v,i}$ be the set of elements in $\texttt{Tasks}_v$ during the first phase of $\mathcal{E}_i$. We use the following notations: U_i stands for $\bigcup_{v\in H_i} T_{v,i}$, S_i for $\bigcap_{v\in H_i} T_{v,i}$, and $u_i = |U_i|$, $s_i = |S_i|$. When an epoch number i is fixed, we drop the subscript i in $T_{v,i}$ and write simply T_v.

4.2 Deterministic balancing algorithm BAL

In this section we consider an algorithm obtained from the generic version of selection algorithms of Section 4.1 by replacing generic selection rule by a specific rule `Load_Balance` to select items by load balancing. This rule operates as follows. If l is the number of elements in the list at hand, then processor $\mathcal{P}_v$ selects the item of rank $r(v)$, where the number $r(v)$ is uniquely determined by inequalities

$$\frac{p}{l}(r(v)-1) < v < \frac{p}{l}(r(v)+1) \ . \tag{4.1}$$

Throughout this section, notation $H_i = H$ means a fixed good subgraph of G which has all the elements active the longest time among all such subgraphs. Such a subgraph exists by PROPERTY $\mathcal{D}$: the algorithm is deterministic, hence the adversary can decide which processors to fail prior to the start of the algorithm, this determines the subgraph G' of operational nodes and a good subgraph $H' \subseteq G'$. Such an H exists because H' is a good subgraph which has all the elements active during the first phase of epoch $\mathcal{E}_1$.

Lemma 4.4 *If $u_i \geq 11p^2g(p)$ then $u_i - u_{i+1} \geq \varrho p g(p)$.*

Proof: It follows from the way the lists are updated that inequalities

$$u_{i+1} \leq s_i \tag{4.2}$$

hold. Suppose first that inequality

$$u_i - s_i \geq pg(p) \tag{4.3}$$

holds. Combine it with inequality (4.2) to obtain

$$\begin{aligned} u_i - u_{i+1} &\geq u_i - s_i \\ &\geq pg(p) \\ &> \varrho p g(p) \ . \end{aligned}$$

Next consider the situation when inequality (4.3) does not hold, that is, $u_i - s_i < pg(p)$. List $\texttt{Task}_v$ can lose at most $u_i - s_i < pg(p)$ items during epoch $\mathcal{E}_i$ due to the discrepancy of the local knowledge of $\mathcal{P}_v$ about the outstanding tasks and the true situation, and also at most $pg(p)$ items performed during $\mathcal{E}_i$. We want to show that during $\mathcal{E}_i$ each task is performed by at most one processor. This will be achieved by considering the ranks of tasks in the ordered set $U_i = \bigcup_{v\in H_i} T_v$. Processor v performs a task whose rank is $r(v)$ in $\texttt{Task}_v$ in the first phase of $\mathcal{E}_i$. Let $w(v)$ be the rank in U_i of this task. Consider any other task performed by $\mathcal{P}_v$ during epoch $\mathcal{E}_i$. Let $r'(v)$ be its rank in $\texttt{Task}_v$ when it was selected and let $w'(v)$ be its rank in U_i. Note that list $\texttt{Task}_v$ is a dynamic data structure, and ranks of its elements may change during $\mathcal{E}_i$, while the rank of a task with respect to U_i remains the same during $\mathcal{E}_i$. We will need inequalities

$$|T_v| \cdot \frac{v}{p} - 1 < r(v) < |T_v| \cdot \frac{v}{p} + 1 \ ,$$

which follow from (4.1). List $\texttt{Task}_v$ may loose at most $2pg(p)$ items during $\mathcal{E}_i$, so we obtain the following inequalities:

$$(|T_v| - 2pg(p)) \cdot \frac{v}{p} - 1 < r'(v) < |T_v| \cdot \frac{v}{p} + 1 \ .$$

Therefore the ranks in U_i satisfy the following pairs of inequalities:

$$|T_v| \cdot \frac{v}{p} - 1 < w(v) < [|T_v| + pg(p)] \cdot \frac{v}{p} + 1 \ ,$$

and

$$(|T_v| - 2pg(p)) \cdot \frac{v}{p} - 1 < w'(v) < [|T_v| + 2pg(p)] \cdot \frac{v}{p} + 1 + 2pg(p) \ .$$

This implies the bound

$$|w'(v) - w(v)| < 3pg(p) + 2 \ . \tag{4.4}$$

For each processor $\mathcal{P}_v$ we consider set $B_v \subseteq U_i$ consisting of the tasks whose ranks differ from $w(v)$ by at most $3pg(p) + 2$. It follows from (4.4) that all the tasks performed by $\mathcal{P}_v$ during $\mathcal{E}_i$ are in B_v. Consider two different processors $\mathcal{P}_{v_1}$ and $\mathcal{P}_{v_2}$, $v_1 > v_2$, we will show that their corresponding sets B_{v_1} and B_{v_2} are disjoint. To this end we will estimate the distance in U_i of the task they perform in the first phase of $\mathcal{E}_i$. We use inequalities

$$r(v_1) > |T_{v_1}| \cdot \frac{v_1}{p} - 1$$

and

$$r(v_2) < |T_{v_2}| \cdot \frac{v_2}{p} + 1$$

for the ranks in lists $\texttt{Task}_v$. They imply the following inequalities concerning the ranks in U_i:

$$w(v_1) > |T_{v_1}| \cdot \frac{v_1}{p} - 1$$

and

$$w(v_2) < |T_{v_2}| \cdot \frac{v_2}{p} + 1 + pg(p) \ .$$

The final estimation is as follows:

$$\begin{aligned}
w(v_1) - w(v_2) &> |T_{v_1}| \cdot \frac{v_1}{p} - 1 - |T_{v_2}| \cdot \frac{v_2}{p} - 1 - pg(p) \\
&> (|T_{v_2}| - pg(p)) \cdot \frac{v_1}{p} - |T_{v_2}| \cdot \frac{v_2}{p} - 2 - pg(p) \\
&> |T_{v_2}| \cdot \frac{v_1 - v_2}{p} - (v_1 + p)g(p) - 2 \\
&> (u_i - pg(p)) \cdot \frac{v_1 - v_2}{p} - (v_1 + p)g(p) - 2 \\
&> 10pg(p) - 2pg(p) - 2 \\
&> 7pg(p)
\end{aligned}$$

where we used the assumption that $u_i \geq 11p^2 g(p)$. Note that $7pg(p) > 2[3pg(p) + 2]$. This shows that the sets B_{v_1} and B_{v_2} are disjoint. $\square$

Lemma 4.5 *Let $u = u_{i_0}$. For $i > i_0$, if during epoch $\mathcal{E}_i$ all the processors in H are in the* MAIN PHASE *and $u_i > \sqrt{\frac{u}{\log u}}$, then $u_i - u_{i+1}$ is either at least*

$$\min(\frac{\varrho}{14}\sqrt{\frac{u}{\log u}}, \varrho p)$$

or at least

$$u_i \cdot \sqrt{\frac{\log u}{u}} \ .$$

Proof: Denote $x = \sqrt{u/\log u}$ to simplify the notation. Suppose

$$u_i - u_{i+1} < u_i/x \ . \tag{4.5}$$

It is sufficient to show that $u_i - u_{i+1}$ is either at least $\varrho x/14$ or at least ϱp. Let $\mathcal{P}_{v_1}$ and $\mathcal{P}_{v_2}$ be two processors in H which perform the same task in the first phase of epoch $\mathcal{E}_i$, where $v_1 \geq v_2$. It is sufficient to prove that (4.5) implies

$$v_1 - v_2 < 7p/x \ , \tag{4.6}$$

because then either $7p > x$, in which case at least ϱp processors perform different tasks, or at least

$$\frac{\varrho p}{2 \cdot 7p/x} = \frac{\varrho x}{14}$$

processors in H perform different tasks.

Note that (4.5) implies

$$s_i \geq u_i(1 - 1/x) \ . \tag{4.7}$$

This is because $u_i - s_i \leq u_i - u_{i+1}$. Combining (4.7) with the estimates

$$s_i \leq |T_{v_1}|, |T_{v_2}| \leq u_i$$

we obtain

$$(1 - 1/x)u_i \leq |T_{v_1}|, |T_{v_2}| \leq u_i \ , \tag{4.8}$$

what yields

$$||T_{v_1}| - |T_{v_2}|| \leq u_i/x \ . \tag{4.9}$$

Inequality (4.9) implies

$$|r(v_1) - r(v_2)| \leq |T_{v_1} \setminus T_{v_2}| + |T_{v_2} \setminus T_{v_1}| \leq u_i/x \ , \tag{4.10}$$

because $r(v_1)$ and $r(v_2)$ are the ranks of the same task in $\texttt{Task}_{v_1}$ and $\texttt{Task}_{v_2}$ respectively. The difference $v_1 - v_2$ can be upper bounded

$$v_1 - v_2 < \frac{p}{|T_{v_1}|}(r(v_1) + 1) - \frac{p}{|T_{v_2}|}(r(v_2) - 1) \tag{4.11}$$

by (4.1). We can bound $v_1 - v_2$ by plugging the estimates given by (4.8), (4.9) and (4.10) into the right-hand side of (4.11):

$$\begin{aligned} v_1 - v_2 &< p[\frac{r(v_2) + u_i/x + 1}{|T_{v_1}|} - \frac{r(v_2) - 1}{|T_{v_1}| + u_i/x}] \\ &= p\frac{r(v_2)u_i/x + (u_i/x)^2 + u_i/x + (u_i/x)|T_{v_1}| + 2|T_{v_1}|}{|T_{v_1}|(|T_{v_1}| + u_i/x)} \\ &\leq p \cdot \frac{u_i^2}{x} \cdot \frac{r(v_2)/u_i + 1/x + 1/u_i + |T_{v_1}|/u_i + 2x|T_{v_1}|/u_i^2}{u_i(1 - 1/x)(u_i(1 - 1/x) + u_i/x)} \ . \end{aligned} \tag{4.12}$$

Note that inequalities

$$1 \leq r(v_2), |T_{v_1}|, x \leq u_i$$

hold. It follows that each of the terms in the nominator of (4.12) is at most 1. We obtain the following upper bound on (4.12):

$$p \cdot \frac{u_i^2}{x} \cdot \frac{6}{u_i^2(1 - 1/x)} \leq \frac{6p}{x - 1} \leq \frac{7p}{x} \ ,$$

for sufficiently large u and p. This completes the proof of inequality (4.6). $\square$

Theorem 4.6 *The work and communication of algorithm* BAL *is*

$$\mathcal{O}(t + p\log p\sqrt{(p+t)(\log p + \log t)})$$

against the adversary LINEARLY-BOUNDED.

Proof: We will estimate the work $\mathcal{W}$, the communication is $\mathcal{O}(\mathcal{W})$. We partition all the numbers of epochs into five disjoint intervals I_j, $1 \le j \le 5$, in such a way that if $a \in I_{j_1}$ and $b \in I_{j_2}$, for $j_1 < j_2$, then $a < b$. The partitioning is as follows. Interval I_1 contains those i for which $u_i \ge 11p^2 g(p)$. Interval I_2 contains those i for which both $u_i < 11p^2 g(p)$ and $s_i \ge 1$. Hence if $i \in I_2$ then no local knowledge of a processor in H implies that all the tasks have been done. Interval I_3 contains just a single number i of the first epoch $\mathcal{E}_i$ following the epochs with numbers in I_2. During $\mathcal{E}_i$ all the processors in H learn that all the tasks are performed. Number i is in interval I_4 if H is still good during the first phase of epoch $\mathcal{E}_i$. Finally interval I_5 contains the remaining numbers of epochs.

We will estimate the sizes of intervals I_j. By Lemma 4.4, during epochs in I_1 all the processors in H perform different tasks, hence

$$|I_1| = \mathcal{O}(t/(pg(p))) \ . \tag{4.13}$$

During I_2 all the processors from H are in the MAIN PHASE. Let i_0 be the minimal element in I_2. Let $x = \sqrt{u/\log u}$, where $u = u_i$. We further partition I_2 into $I_2 = I_2^1 \cup I_2^2 \cup I_2^3$ as follows. Number $i \in I_2^1$ iff $u_i \le x$. Let the remaining numbers of epochs in I_2 satisfy the following: if $i \in I_2^2$ then $u_i - u_{i+1} \ge u_i/x$, and if $i \in I_2^3$ then $u_i - u_{i+1} \ge \min(\varrho x/14, \varrho p)$. Lemma 4.5 guarantees that such a partition of I_2 is possible. We estimate the sizes of components of I_2:

$$|I_2^1| = \mathcal{O}(x/g(p)) = \mathcal{O}(\sqrt{t/\log t}/g(p)) \ .$$

If $i \in I_2^2$ then $u_{i+1} \le u_i(1 - 1/x)$ so the size of I_2^2 is $\mathcal{O}(\sqrt{u\log u}) = \mathcal{O}(\sqrt{t\log t})$, because

$$u(1-1/x)^{x\log u} = \mathcal{O}(1) \ .$$

The size of I_2^3 is

$$\begin{aligned}\mathcal{O}(\max(u/x, u/p)) &= \mathcal{O}(\sqrt{u\log u} + u/p)\\ &= \mathcal{O}(\sqrt{t\log t} + \sqrt{ug(p)})\\ &= \mathcal{O}(\sqrt{t(g(p)+\log t)})\end{aligned}$$

because $u < 11p^2 g(p)$. It follows that

$$|I_2| = \mathcal{O}(\sqrt{t(g(p)+\log t)}) \ . \tag{4.14}$$

During I_4 all the processors in H are in the CLOSING PHASE. An analysis similar to that of I_2 applies, when the items in lists $\mathtt{Busy}_v$ are treated as tasks, so u is at most p. We obtain

$$|I_4| = \mathcal{O}(\sqrt{p(g(p) + \log p)}) . \tag{4.15}$$

Next we show that $|I_5| = \mathcal{O}(1)$. To this end observe that during the first epoch in interval I_5 there is no good processor. Hence during the next epoch all the active processors halt.

The whole work can be estimated by

$$\mathcal{O}\big(pg(p)\sum_{j=1}^{5} |I_j|\big) ,$$

which yields the claimed bound by the estimates (4.13), (4.14) and (4.15). $\square$

4.3 Randomized algorithm RAL

Randomized algorithm RAL is obtained from generic scheme of selection algorithms by applying rule `Select_Randomly` to choose items from lists. This rule operates in the same way as `Load_Balance` while the size of the list at hand is at least $11p^2g(p)$, otherwise an item is selected randomly, and selections at different steps and of different processors are independent. Note that graphs G_i induced by active nodes during the first phase of epoch $\mathcal{E}_i$ cannot be fixed in advance, since the adversary is failing nodes depending on random decisions of the processors to select items from their lists. Therefore we need to resort to PROPERTY $\mathcal{R}$ to define H_i, as explained in Section 4.1.1. We say that an event happens *with high probability* (`whp`) if its probability to hold is at least $1 - \exp(-\alpha p)$, for a constant $\alpha > 0$.

Lemma 4.7 *If $\varrho pg(p)/2^5 \le u_i < 11p^2g(p)$ then inequality*

$$u_i - u_{i+1} \ge \lambda pg(p)$$

holds `whp`*, for a constants $\lambda > 0$.*

Proof: Let us fix $R \subseteq H_i$ and $|R| = \varrho p$. Each processor $\mathcal{P}_v$ in R selects randomly a subset of $g(p)$ tasks in T_v. We want to show that processors in R perform at least $\lambda pg(p)$ tasks `whp`, for a constant $\lambda > 0$. This can be modeled by placing balls in urns: $\varrho pg(p)$ balls are put randomly into at least $\varrho pg(p)$ urns. The Chernoff bound (cf. [4]) gives the probability estimate $1 - \exp(-\Theta(pg(p)))$. There are

$$\binom{p}{\varrho p} \le 2^{ph(\varrho)}$$

such subsets R. We may sum over all such subsets R to obtain a bound of the form

$$1 - 2^{ph(\varrho)} \exp(-\Theta(pg(p))) \geq 1 - \exp(-\Theta(p)) \ ,$$

because $g(p) = \gamma \cdot \lg p$ diverges to infinity with $p \to \infty$. □

Lemma 4.8 *Suppose inequalities $2\varrho p \leq u_i < \varrho p g(p)/2^5$ hold. Then,* `whp`*, either inequality*

$$u_{i+1}/u_i \leq u_i/\varrho p g(p)$$

holds or at least $\varrho p/3$ processors fail during epoch $\mathcal{E}_i$.

Proof: Let $W \subseteq \sum_{v\in H_i} T_v$ be an arbitrary subset of size $u_i^2/\varrho p g(p)$. Let $\mathcal{P}_v$ be among the first ϱp processors in H_i. The probability that $\mathcal{P}_v$ does not perform a task from W during epoch $\mathcal{E}_i$ is at most

$$\begin{aligned}\big(1 - \frac{|W|}{u_i}\big)^{g(p)} &\leq \exp\big(-\frac{|W|}{u_i}\cdot g(p)\big)\\ &= \exp\big(-\frac{u_i}{\varrho p}\big) \ .\end{aligned}$$

Given $\mathcal{P}_v \in H_i$, let the random variable X_v have value 1 if $\mathcal{P}_v$ does not perform a task in W during $\mathcal{E}_i$, and 0 otherwise. Define $X = \sum X_v$, where we sum over the first ϱp processors $\mathcal{P}_v$ in H_i. We will need to estimate the deviation of X from its expected value $\mathsf{E}[X]$. To do this, consider a sequence of independent Bernoulli trials $Y_i, \ldots, Y_{\varrho p}$, where $\mathsf{P}[Y_v = 1] = \exp(-u_i/\varrho p)$. Let $Y = \sum Y_v$, and $\mu = \mathsf{E}[Y] = \varrho p \exp(-u_i/\varrho p)$. By the Chernoff bound we have

$$\mathsf{P}[Y \geq b\cdot\mu] \leq \exp\big(\mu(b - 1 - b\ln b)\big) \ . \tag{4.16}$$

Choose $b = (2/3)\exp(u_i/\varrho p)$. After substituting the values of b and μ into (4.16) we obtain

$$\begin{aligned}\mathsf{P}[Y \geq \frac{2}{3}\varrho p] &\leq \exp\Big(\varrho p \exp\big(-\frac{u_i}{\varrho p}\big)\big((2/3)\exp\big(\frac{u_i}{\varrho p}\big) - 1 - (2/3)\frac{u_i}{\varrho p}\exp\big(\frac{u_i}{\varrho p}\big)\big)\Big)\\ &\leq \exp\Big(-\varrho p\exp\big(-\frac{u_i}{\varrho p}\big)\big((1/3)\frac{u_i}{\varrho p}\exp\big(\frac{u_i}{\varrho p}\big)\big)\Big)\\ &\leq \exp(-u_i/3) \ ,\end{aligned} \tag{4.17}$$

where we used inequality $u_i \geq 2\varrho p$. We have

$$\mathsf{P}[X \geq 2\varrho p/3] \leq \mathsf{P}[Y \geq 2\varrho p/3] \ ,$$

because $\mathsf{P}[X_v = 1] \le \mathsf{P}[Y_v = 1]$. The number of all such subsets W is

$$\begin{aligned} \binom{u_i}{u_i^2/\varrho p g(p)} &\le 2^{u_i h(u_i/\varrho p g(p))} \\ &\le 2^{u_i h(2^{-5})} \ , \end{aligned} \tag{4.18}$$

where

$$h(x) = -[x \lg x + (1-x)\lg(1-x)]$$

is the (binary) entropy function. We are interested in the event that for each subset W of size $u_i^2/\varrho p g(p)$ at least $\varrho p/3$ processors perform each some task in W. It follows from (4.17) and(4.18) that the probability that this happens is at least

$$\begin{aligned} 1 - \exp(-u_i/3)\exp\big(u_i h(u_i/\varrho p g(p)) \ln 2\big) &\ge 1 - \exp\big((u_i/3)(-1+\ln 2)\big) \\ &\ge 1 - \exp(-\alpha u_i) \ , \end{aligned}$$

for a constant $\alpha > 0$, because $h(2^{-5}) < 1/3$. We have shown that `whp` the only possibility for the adversary to preserve $u_{i+1} > u_i^2/\varrho p g(p)$ tasks for the next epoch $\mathcal{E}_{i+1}$ is to fail at least $\varrho p/3$ processors. □

Lemma 4.9 *If $u_i < 2\varrho p$ then either inequality $u_{i+1} \le u_i/g(p)$ holds* `whp` *or $\Omega(p)$ processors fail* `whp`.

Proof: Note that all the processors in H_i are in the MAIN PHASE. Let W be a subset of U_i of size $u_i/g(p)$. First we show that for some constant $\lambda > 0$ the size of the set of processors in H_i which perform a task in W is at least λp `whp`. Fix one processor $\mathcal{P}_v$. The probability that $\mathcal{P}_v$ will not perform a task in W is at most

$$\begin{aligned} \big(1 - \frac{|W|}{u_i}\big)^{g(p)} &= \big(1 - \frac{1}{g(p)}\big)^{g(p)} \\ &< 1/e \ . \end{aligned}$$

Let X_v be the random variable equal to 1 if $\mathcal{P}_v$ performs a task in W, and 0 otherwise. The inequality $\mathsf{P}[X_v = 1] > 1/e$ holds, hence $\sum_{v \in H_i} X_v = \Omega(p)$ with the probability at least $1 - \exp(-\beta p)$ by the Chernoff bound, for some constant $\beta > 0$. Suppose, to the contrary, that

$$u_{i+1} > u_i/g(p) \ .$$

Hence there is a set W of size $u_i/g(p)$ such that none among the elements in W is performed. The number of such subsets is

$$\begin{aligned} \binom{u_i}{u_i/g(p)} &\le 2^{u_i h(1/g(p))} \\ &\le 2^{2\varrho p h(1/g(p))} \ . \end{aligned}$$

We are interested in the event that for each subset of tasks of size $u_i/g(p)$ there are some λp processors such that each of them performs a task in W, where $\lambda > 0$ is a constant. It follows that this event holds with the probability at least

$$1 - \exp(-\beta p) \cdot 2^{2\varrho p h(1/g(p))} \geq 1 - \exp(-\alpha p) \ ,$$

for a certain constant $\alpha > 0$, because $h(x)$ converges to zero when x tends to zero, and the function $g(p) = \gamma \cdot \lg p$ gets arbitrarily large as p grows to infinity. If this happens then the only possibility for the adversary to preserve more than $u_i/g(p)$ tasks for epoch $\mathcal{E}_{i+1}$ is to fail $\Omega(p)$ processors. □

Theorem 4.10 *The work and communication of algorithm* RAL *is*

$$\Theta\big(t + p\ \frac{\log^2 p}{\log\log p}\big)$$

`whp` *against the adversary* Strongly-Adaptive Linearly-Bounded.

Proof: We show that the algorithm terminates after

$$\mathcal{O}\big(t/(pg(p)) + \log p/\log g(p)\big)$$

epochs with a large probability. We partition all the numbers of epochs into disjoint intervals I_j, similarly as in the proof of Lemma 4.6. The estimation

$$|I_1| = \mathcal{O}(t/(pg(p)))$$

holds by Lemma 4.4. Let us partition I_2 into $I_2^1 \cup I_2^2 \cup I_2^3$, where $i \in I_2^1$ iff $\varepsilon\varrho pg(p) \leq u_i < 11p^2g(p)$, $i \in I_2^2$ iff $2\varrho p \leq u_i < \varepsilon\varrho pg(p)$, and $i \in I_2^3$ iff $u_i < 2\varrho p$. By Lemma 4.7, the size of subinterval I_2^1 is $\mathcal{O}(t/(pg(p)))$ with the probability at least

$$1 - \Theta(p)\exp(-\alpha p) \geq 1 - \exp(-\beta p) \ ,$$

for certain constants $\alpha, \beta > 0$. We will estimate the size of subinterval I_2^2 by using Lemma 4.8. Note that there are $\mathcal{O}(1)$ epochs during which the adversary may fail $\Omega(p)$ processors, hence we may assume without loss of generality that inequalities $u_{i+1} \leq u_i^2/\varrho pg(p)$ hold `whp` for each $i \in I_2^2$. Let k be the largest epoch number in I_2^2, and let $a = |I_2^2|$. Unwinding inequality $u_{i+1} \leq u_i^2/\varrho pg(p)$ we obtain the following sequence of inequalities:

$$
\begin{aligned}
2\varrho p &\le u_k \\
&\le \frac{u_{k-1}^2}{\varrho p g(p)} \\
&\le \frac{u_{k-2}^4}{\big(\varrho p g(p)\big)^3} \\
&\le \dots \\
&\le \frac{u_{k-a+1}^{2^{a-1}}}{\big(\varrho p g(p)\big)^{2^{a-1}-1}} \ .
\end{aligned}
$$

Since $u_{k-a+1} \le \varrho p g(p)/2^5$, we denote $\varepsilon = 2^{-5}$ and obtain inequalities

$$
\begin{aligned}
2\varrho p &\le \frac{\big(\varepsilon\, \varrho p g(p)\big)^{2^{a-1}}}{\big(\varrho p g(p)\big)^{2^{a-1}-1}} \\
&\le \varrho p g(p) \cdot \varepsilon^{2^{a-1}} \ .
\end{aligned}
$$

It follows that inequality

$$2^{2^{(a-1)\log_2 5}} \le g(p)/2$$

holds and we obtain $|I_2^2| = a = \mathcal{O}\big(\log\log g(p)\big)$ `whp`.

Lemma 4.9 can be applied for the epochs from I_2^3. The number of epochs during which $\Omega(p)$ processors fail is $\mathcal{O}(1)$, so we may assume that inequalities $u_{i+1} \le u_i/g(p)$ hold `whp` for each $i \in I_2^3$. We obtain that the size of I_2^3 is

$$\mathcal{O}\big(\log(pg(p))/\log g(p)\big) = \mathcal{O}\big(\log p/\log g(p)\big) \ ,$$

because

$$pg(p) \cdot g(p)^{-\log(pg(p))/\log g(p)} = \mathcal{O}(1)$$

and $g(p) = \mathcal{O}(p)$. This holds with the probability

$$1 - \mathcal{O}(\log p)\exp(-\alpha p) \ge 1 - \exp(-\beta p) \ ,$$

for certain constants $\alpha, \beta > 0$. It follows that `whp`

$$|I_2| = \mathcal{O}\big(t/(pg(p)) + \log p/\log g(p)\big) \ .$$

We have $|I_3| = \mathcal{O}(1)$. To estimate the size of I_4, replace t by p in the bound on $|I_2|$. We obtain that the estimation

$$|I_4| = \mathcal{O}(\log p/\log g(p))$$

holds `whp`. Finally, $|I_5| = \mathcal{O}(1)$ `whp`: this is because all the processors from the last H_i cease to be active during $\mathcal{O}(1)$ epochs, and the number of sequences of H_i is also $\mathcal{O}(1)$. We during that `whp` the following estimate holds:

$$\sum_{j=1}^{5} |I_j| = \mathcal{O}\left(\frac{t}{pg(p)} + \frac{\log p}{\log g(p)}\right) ,$$

what completes the proof. □

4.4 Permutation algorithms

In this section we explore another paradigm in which processors select items from lists in the order given by their private permutations. This defines a family of selection algorithms that we call *permutation algorithms*. We consider randomized algorithm RPAL and a scheme of deterministic algorithms DPAL parametrized by families of permutations.

Permutation algorithms are determined by selection rule `Select_According_-To_Permutations` which operates as follows. Suppose each processor $\mathcal{P}_v$ has its own two permutations, one of the numbers in $[1..p]$ and the other of $[1..11p^2g(p)]$. Processor $\mathcal{P}_v$ starts by permuting the list $\texttt{Busy}_v$ according to the permutation of $[1..p]$. If it needs to select an item from $\texttt{Busy}_v$ then it takes the first one in the list. Tasks are selected the same way as in rule `Load_Balance` as long as the size of $\texttt{Tasks}_v$ is greater than $11p^2g(p)$. At the moment when exactly $11p^2g(p)$ items remain in $\texttt{Tasks}_v$, processor $\mathcal{P}_v$ permutes the list $\texttt{Tasks}_v$ according to the permutation of $[1..11p^2g(p)]$. Then, at any time when $\mathcal{P}_v$ needs to select an item from $\texttt{Tasks}_v$, it simply takes the first one in the list.

To fully specify a permutation algorithm, we need permutations for each processor. We consider two scenarios. Randomized algorithm RPAL starts by each processor selecting two permutations at random, independently among all the processors. Deterministic algorithm DPAL has a fixed pair of permutations assigned to each processor. We analyze the expected behavior of algorithm RPAL and apply this analysis by way of the probabilistic method to show that the algorithm DPAL is efficient for many families of pairs of permutations. Deterministic algorithm DPAL is more efficient than the deterministic algorithm BAL, however the implementation of BAL is straightforward, whereas the development of DPAL is not constructive.

Randomized algorithm RAL has a better expected upper bound on its work and communication than the one we prove in this section for RPAL. This is because processors performing RAL make random choices one at a time and at the latest possible moment, which gives the adversary only a probability distribution on the future choices. The processors running RPAL select their random permutations at the start of the algorithm, and when this has been completed then the algorithm becomes deterministic and the adversary has a perfect knowledge about the future.

Algorithm RPAL has a larger upper bound on the expected work and communication, as compared to RAL, but this bound is proved with better probability estimates. Namely, the upper bound on the work and communication of RAL holds `whp`, whereas the upper bound proved for RPAL holds with even higher probability defined as follows.

We say that an event holds *with very large probability* (`wvlp`) if the respected probability is at least $1-\exp(-\alpha p \log p)$, for a constant $\alpha > 0$. Once the permutation has been selected in an execution of RPAL, the future behavior of the adversary is determined by a *failure pattern* F, which is a subset R of processors of size at most $(1-c)p$, and for each processor $\mathcal{P}_v \in R$ the time step during which $\mathcal{P}_v$ fails. In this section H denotes a good subgraph $H \subseteq G$ of size ϱp. The notation $\mathcal{F}_H$ means the set of all the failure patterns which do not fail any node in H.

Lemma 4.11 *Suppose the adversary can select only the failure patterns from $\mathcal{F}_H$. If $u_i \geq a$ then inequality $u_i - u_{i+1} \geq (1/3)\min\{a, \varrho p g(p)\}$ holds* `wvlp`.

Proof: Let $b = (1/3)\min\{a, \varrho p g(p)\}$. By inequality $u_{i+1} \leq s_i$ we need to consider only the case $s_i \geq 2u_i/3$. Consider a conceptual process in which the processors in H perform the tasks sequentially, the next processor takes over when the previous one has performed all its tasks during epoch $\mathcal{E}_{i+1}$. Let U_i^k denote the set of tasks from U_i which have not been performed during the first k rounds of the processor. Define the random variables X_k, for $1 \leq k \leq \varrho p g(p)$, as follows: if either $u_i - |U_i^k| \geq b$ or $U_i^k \neq U_i^{k-1}$ then $X_k = 1$, otherwise $X_k = 0$.

Note that $\mathsf{P}[X_k = 1] \geq 1/3$. Namely, let processor $\mathcal{P}_v$ is to perform round k. If $u_i - |U_i^k| < b$ then we also have the following inequalities

$$\begin{aligned} s_i - \big(u_i - |U_i^k|\big) &> s_i - b \\ &\geq u_i/3 \\ &\geq |T_v|/3 , \end{aligned}$$

because $u_i \geq a \geq 3b$. This means that at least one third of the tasks in $\texttt{Tasks}_v$ are still not performed by the processors from H.

We need to estimate the probability $\mathsf{P}[\sum X_k \geq b]$. Consider a sequence Y_k of independent Bernoulli trials, with $\mathsf{P}[Y_k = 1] = 1/3$. Then

$$\mathsf{P}[\sum X_k \geq d] \geq \mathsf{P}[\sum Y_k \geq d]$$

for any $d > 0$. Since $b \leq \varrho p g(p)/3$, the result follows by the Chernoff bound. $\Box$

Lemma 4.12 *If the adversary can select only the failure patterns from $\mathcal{F}_H$ then the work and communication of* RPAL *is* $\mathcal{O}(t + p\log^2 p)$ `wvlp`.

Proof: Since the communication is proportional to work, it is sufficient to show that the number of epochs is $\mathcal{O}\big(t/(pg(p)) + \log p\big)$ `wvlp`.

We partition all the numbers of epochs into disjoint intervals I_j, similarly as in the proof of Lemma 4.6. The estimate

$$|I_1| = \mathcal{O}(t/(pg(p)))$$

is by Lemma 4.4. Let us partition I_2 into $I_2^1 \cup I_2^2$, where $i \in I_2^1$ iff $\varrho p g(p) \leq u_i < 11p^2 g(p)$, if $i \in I_2^2$ then $u_i < \varrho p g(p)$. By Lemma 4.11, with $a = \varrho p g(p)$, the size of I_2^1 is $\mathcal{O}(t/(pg(p)))$ with the probability at least

$$1 - p\exp(-\alpha p g(p)) \geq 1 - \exp(-\beta p g(p)) \ ,$$

for certain constants $\alpha, \beta > 0$. By Lemma 4.11, with $a = u_i$, we obtain $|I_2^2| = \mathcal{O}(\log(pg(p))$ `wvlp`, because

$$pg(p) \cdot \big(1/3\big)^{\mathcal{O}(\log(pg(p)))} = \mathcal{O}(1) \ .$$

It follows that the estimate

$$|I_2| = \mathcal{O}\big(t/(pg(p)) + \log p\big)$$

holds `wvlp`. Clearly $|I_3| = \mathcal{O}(1)$. To estimate the size of I_4, replace t by p in the bound on $|I_2|$. We obtain that

$$|I_4| = \mathcal{O}(\log p)$$

holds `wvlp`. Also $|I_5| = \mathcal{O}(1)$ `wvlp`. Hence finally we obtain that `wvlp` the following estimate holds:

$$\sum_{j=1}^{5} |I_j| = \mathcal{O}\big(\frac{t}{pg(p)} + \log p\big) \ ,$$

what completes the proof. $\Box$

Lemma 4.13 *The work and communication of algorithm* RPAL *is*

$$\mathcal{O}(t + p\log^2 p)$$

`wvlp` *against the adversary* Strongly-Adaptive Linearly-Bounded.

Proof: The estimates on the work and communication are the same as in Lemma 4.12. The difference is that the adversary is not restricted to a specific good subgraph H. Whatever the behaviour of the adversary is, we have by PROPERTY $\mathcal{R}$ that there is a nested sequence of good subgraphs H_i. Let H be included in the intersection of all these subgraphs, H being of size ϱp, also H is good in G. The number of such subgraphs H is at most

$$\binom{p}{\varrho p} \leq 2^{h(\varrho)\cdot p} .$$

Let $\alpha > 0$ be a constant such that each H contributes the probability $\exp(-\alpha p g(p))$ by Lemma 4.12. Summing over all subgraphs H we obtain that the bound on the work and communication holds with the probability at least

$$1 - \exp(ph(\varrho)\ln 2) \cdot \exp(-\alpha p g(p)) \geq 1 - \exp(-\beta p \log p)$$

for a certain constant $\beta > 0$, if only p is sufficiently large, because $g(p) = \gamma \cdot \lg p$ grows to infinity with p. □

Theorem 4.14 states that for certain fixed permutations the work and communication of the respective permutation algorithm is $\mathcal{O}(t + p \cdot \log^2 p)$ against the adversary LINEARLY-BOUNDED. To show this we apply the probabilistic method [4].

Theorem 4.14 *There is a reliable deterministic permutation algorithm* DPAL *which solves the Do-All problem with work and communication* $\mathcal{O}(t + p\log^2 p)$ *against the* LINEARLY-BOUNDED *adversary.*

Proof: A proof is straightforward with the probabilistic analysis at our disposal. The key property of algorithm RPAL is that random permutations are selected at the start. One this has been done the algorithms operates in a fully deterministic manner, and the adversary has full information.

It follows from Lemma 4.13 that a randomly selected family of permutations yields the required performance with a large probability. This implies in particular that there *exists* such a family of permutations. Let us select and fix such a family: this yields deterministic algorithm DPAL. □

Chapter 5

Broadcast Networks

In this chapter we consider the problem *Do-All* in broadcast networks.

Model of communication: multiple-access channel

Collision detection: model dependent

Adversaries: size bounded

We start with a lower bound on reliable algorithms solving *Do-All*. Then in Section 5.1 we develop algorithm TwoLists which is optimal among all reliable algorithms against strongly adaptive adversaries in a channel without collision detection. The next algorithm GroupsTogether is optimal among all reliable algorithms in the channel with collision detection. It is given in Section 5.2 and is an adaptation of TwoLists. Both these algorithms are deterministic. In Section 5.3 we develop a algorithm TwoLists which is shown to be provably optimal against the adversary Weakly-Adaptive Linearly-Bounded. If the number of operational stations can be asymptotically smaller than p then optimality of this algorithm is not settled in this work.

By an *execution* of algorithm we mean a sequence of configurations of the system in consecutive rounds, including the failure pattern, the messages broadcast on the channel, and the sequence of random bits used by each station ([25]). If an execution E' is obtained by modifying the actions of an adversary on some other execution E, then it is assumed that each station in E' receives exactly the same sequence of random bits as in E. If a reliable algorithm is run then no station halts when there are still tasks that have not been completed yet: the remaining stations may be killed and the halted station will not perform any more tasks. This property can be strengthened quantitatively as follows:

Lemma 5.1 *Any reliable algorithm, also possibly randomized, has to perform work*

$\Omega(t + p\sqrt{t})$ in each execution, even if no failures happen.

Proof: It is sufficient to consider the channel with collision detection, in which algorithms have more information. Part $\Omega(t)$ of the bound follows from the fact that each task has to be performed at least once.

A task α is *confirmed* at round i, in an execution of the algorithm, if either a station broadcasts successfully and it has performed α by round i, or more than one station broadcast simultaneously and all of them, with a possible exception of one station, have performed α by round i. At least half of the stations broadcasting in round i and confirming α have performed it by then, so at most $2i$ tasks can be confirmed in round i. Let E_1 be an execution of the algorithm when no failures happen. Let station $\mathcal{P}$ come to a halt at some round j in E_1.

Claim: the tasks not confirmed by round j were performed by $\mathcal{P}$ itself in E_1.

Suppose, on the contrary, that this is not the case, let β be such a task. Consider an execution E_2 obtained by running the algorithm and killing any station that performed β in E_1 just before it was to perform β, and all the remaining stations, except for $\mathcal{P}$, killed at round j. The broadcasts on the channel are the same during the first j rounds in E_1 and E_2. Hence all the stations perform the same tasks in E_1 and E_2 till round j. The definition of E_2 is consistent with the power of the adversary Unbounded. The algorithm is not reliable because task β is not performed in E_2 and station $\mathcal{P}$ is operational. This justifies the claim.

We estimate the contribution of station $\mathcal{P}$ to work: The total number of tasks confirmed in E_1 is at most $2(1 + 2 + \ldots + j) = \mathcal{O}(j^2)$. Suppose some t' tasks have been confirmed by round j. The remaining $t - t'$ tasks have been performed by $\mathcal{P}$. The work of $\mathcal{P}$ is at least $\Omega(\sqrt{t'} + (t - t')) = \Omega(\sqrt{t})$. □

The amount of work asymptotically equal to $t + p\sqrt{t}$ is called *minimum*. Lemma 5.1 shows that the minimum work is an absolute lower bound on the amount of work performed by a reliable algorithm in any scenario. The minimum work is a yardstick we will use in the following sections to measure the performance of algorithms.

5.1 Channel without collision detection

We develop a deterministic algorithm TwoLists. At each communication time step there is at most one station scheduled to broadcast, hence the algorithm avoids conflicts between broadcasting stations. A broadcast message may consist of just a single bit, since its only purpose is to confirm that the station is still alive. A station does not need to announce which tasks it performed, since all the stations can compute this themselves. The stations have the same global knowledge implied by what was broadcast on the channel.

Each station $\mathcal{P}_v$ maintains four circular lists:

$\texttt{TASKS}_v$, $\texttt{DONE}_v$, $\texttt{LOCALTASKS}_v$, $\texttt{STATIONS}_v$.

The items in $\texttt{TASKS}_v$ are the tasks still not announced on the channel as performed. The items on list $\texttt{DONE}_v$ are those which have been performed by v during rounds with number 2. The items of $\texttt{LOCALTASKS}_v$ are on list $\texttt{Task}_v$ and have not been performed by station $\mathcal{P}_v$ yet. Finally the items in $\texttt{STATIONS}_v$ are the stations which either made a broadcast each time they were to do so or were not scheduled to broadcast yet at all. Since lists $\texttt{TASKS}_v$ and $\texttt{STATIONS}_v$ are defined by what happened on the channel, they are exactly the same in every station. The lists $\texttt{DONE}_v$ and $\texttt{LOCALTASKS}_v$ are local, but can be computed by another stations in any moment.

There is a pointer associated with each of the lists. Let $\texttt{Station}_v$ denote the station pointed on the list $\texttt{STATIONS}_v$ and $\texttt{Task}_v$ be the task pointed on the list $\texttt{TASKS}_v$. The pointers provide reference points to assign tasks to stations: $\texttt{Task}_v$ is assigned to $\texttt{Station}_v$, then the next task on $\texttt{TASKS}_v$ is assigned to the next station on $\texttt{STATIONS}_v$, and so on in the circular order. Note that a task can be assigned to a number of stations if the list $\texttt{STATIONS}_v$ winds around the list $\texttt{TASKS}_v$ during the assignment processor.

After a scheduled broadcast one of the lists is updated as follows. If a successful broadcast happened then the tasks which have been performed by the broadcasting station are removed from $\texttt{TASKS}_v$. If a broadcast did not happen then the station which failed to broadcast is removed from the list $\texttt{STATIONS}_v$. If an item pointed at by the pointer is removed from a list then the pointer is assigned automatically to the next item on the list. Other then that the pointers are updated as follows: pointer $\texttt{Task}_v$ is moved by $\lceil\sqrt{\texttt{Shift}_v}\rceil$ positions, and pointer $\texttt{Station}_v$ is always advanced by one position.

The variable $\texttt{Shift}_v$ is initialized to value $\lceil t/2\rceil$. The lists $\texttt{TASKS}_v$ and $\texttt{STATIONS}_v$ are initialized to contain all the tasks and stations, respectively, ordered by their identifiers. The size of $\texttt{LIST}$ is denoted as $|\texttt{LIST}|$. Variable $\texttt{Counter}_v$ is initialized to number t. Note that variables $\texttt{Shift}_v$ and $\texttt{Counter}_v$ are exactly the same in every station. Algorithm TwoLists is in phases repeated in a loop, it is described in Figure 5.1. A phase is given in Figure 5.2.

Lemma 5.2 *Algorithm* TwoLists *is reliable.*

Proof: We prove this fact in a sequence of statements:

1) `Shift` $:= \lceil t/2 \rceil$
2) $\texttt{Counter}_v := t$
3) $\texttt{DONE}_v := \emptyset$
4) $\texttt{LOCALTASKS}_v :=$`TASKS`
5) initialize $\texttt{Station}_v$ to the first element of $\texttt{STATIONS}_v$
6) initialize $\texttt{Task}_v$ to the first element of $\texttt{TASKS}_v$

while not halted **do execute** TwoListsPhase

Figure 5.1: Algorithm TwoLists for station $\mathcal{P}_v$.

1. Operational stations have the same local knowledge except possibly `LOCALTASKS`.

   By induction.

2. Each station halts eventually.

   The counter is decreased in each phase.

3. During each phase list `LOCALTASKS` shrinks.

   It follows from the definition of Round 1, and inequality $|\texttt{LOCALTASKS}| \leq$`Counter` which follows by induction.

4. If a task is removed from list `TASKS` then it has been performed.

   This happens only when a processor that performed this task broadcast this information on the channel.

5. All the tasks are performed if at least one station remains operational.

   It follows from the observation that a station can halt either if `Counter`$= 0$ or `TASKS` is empty. Statements 3. and 4. imply that all the tasks have been performed at the moment when the first station halts voluntarily. $\square$

The next theorem shows how the performance of TwoLists degrades gracefully with the number of faults.

Theorem 5.3 *Algorithm* TwoLists *solves Do-All with work* $\mathcal{O}(t+p\sqrt{t}+p\cdot\min\{f,t\})$ *against the adversary* f-Bounded, *for* $0 \leq f < p$.

Proof: The time of computation is partitioned into *epochs*, during an epoch the value of `Shift` is constant.

ROUND 1:

Step 1: pause

Step 2:

- perform the first task after $\texttt{Task}_v$ not performed by $\mathcal{P}_v$ yet and remove it from $\texttt{LOCALTASKS}_v$
- decrease $\texttt{Counter}_v$ by 1
- if $\texttt{Counter}_v \leq 0$ then halt

ROUND 2:

Step 1: if $\texttt{Station}_v = v$ then broadcast one bit

Step 2: attempt to receive a message and do the following:

- if a broadcast is heard then
  a) compute the current value of list $\texttt{DONE}_{\texttt{Station}}$ as maintained by $\texttt{Station}_v$ and remove its elements from $\texttt{TASKS}_v$
  b) if $\texttt{TASKS}_v$ is empty then halt
  c) advance the pointer on $\texttt{STATIONS}_v$ by one position

  else remove $\texttt{Station}_v$ from $\texttt{STATIONS}_v$
- if $|\texttt{TASKS}_v| < \texttt{Shift}_v$ then $\texttt{Shift}_v := \lceil \texttt{Shift}_v/2 \rceil$
- advance the pointer on $\texttt{TASKS}_v$ by $\lceil \sqrt{\texttt{Shift}_v} \rceil$ positions
- perform the assigned task and add it to list $\texttt{DONE}_v$

Figure 5.2: TWOLISTSPHASE for station $\mathcal{P}_v$.

If no broadcast is heard at a step then both the work performed by all the stations at this step and the work performed by the station that was to broadcast and counted since its last successful broadcast is at most $2p$. There are $\mathcal{O}(\min\{f,t\})$ such epochs, because a station performs a new task in each phase, unless failed.

From now we consider only the work performed during rounds with number 2 in a phase. Let us count the work performed by broadcasting stations during an epoch.

To asses this work performed while $|\texttt{STATIONS}_v|^2 \leq \texttt{Shift}$ note that each task performed and reported was performed only once in that way, and this can be charged to $\mathcal{O}(t)$.

Consider next the work performed while $|\texttt{STATIONS}_v|^2 > \texttt{Shift}_v$. The segment of consecutive items on $\texttt{STATIONS}_v$ of size $\lceil\sqrt{\texttt{Shift}_v}\rceil$, starting from $\texttt{Station}_v$, perform different tasks in a phase. Hence, by definition of $\texttt{Shift}_v$ and epoch, each station, except possibly for the first $\mathcal{O}(\sqrt{\texttt{Shift}_v})$, makes the list $\texttt{TASKS}_v$ shorter by at least $\mathcal{O}(\sqrt{\texttt{Shift}_v})$ elements after successful broadcast. Each of these stations is eventually heard, if not failed, so this work can be charged to $\mathcal{O}(p\sqrt{\texttt{Shift}_v})$, since at most $\mathcal{O}(\sqrt{\texttt{Shift}_v})$ stations can broadcast successfully in the epoch. Otherwise the size of $\texttt{TASKS}_v$ list can shorten more than twice which contradicts the definition of epoch.

The values of $\texttt{Shift}_v$ make a geometrically decreasing sequence, hence summing over all epochs gives the estimate $\mathcal{O}(p\sqrt{t})$. □

Next we show a matching lower bound.

Theorem 5.4 *The adversary f-*Bounded*, for $0 \leq f < p$, can force any reliable randomized algorithm for the channel without collision detection to perform work*

$$\Omega(t + p\sqrt{t} + p \cdot \min\{f,t\}) \ .$$

Proof: We consider two cases, depending on which term dominates the bound. If it is $\Omega(t + p\sqrt{t})$ then the bound follows from Lemma 5.1. Consider the case when $\Omega(p \cdot \min\{f,t\})$ is the bound. Let $g = \min\{f,t\}$.

Let E_1 be an execution obtained by running the algorithm and killing any station that wants to broadcast successfully during the first $g/4$ rounds. Denote as A the set of stations failed in E_1. The definition of E_1 is consistent with the power of the adversary f-Bounded, since $|A| \leq g/4 \leq f$.

Claim: no station halts by round $g/4$ in the execution E_1.

Suppose, on the contrary, that some station $\mathcal{P}$ halts before round $g/4$ in E_1. We show that algorithm is not reliable. To this end we consider another execution E_2, which can be realized by the adversary UNBOUNDED. Let γ be a task which is performed in E_1 by at most $pg/(4(t-g/4)) \leq pg/(3t)$ stations except $\mathcal{P}$ during the first $g/4$ rounds. It exists because $g \leq t$. Let B be this set of stations, we have $|B| \leq pg/(3t) \leq p/3$. We define operationally a set C of stations as follows. Initially C equals $A \cup B$; notice $|A \cup B| \leq 7p/12$. If there is any station that wants to broadcast during the first $g/4$ rounds in E_1 as the only station not in the current C then it is added to C. At most one station is added to C for each among the first $g/4 \leq p/4$ rounds of E_1, so $|C| \leq 10p/12 < p$.

Let an execution E_2 be obtained by failing all the stations in C at the start and then running the algorithm. The definition of E_2 is consistent with the power of the adversary UNBOUNDED. There is no broadcast heard in E_2 during the first $g/4$ rounds. Therefore each station operational in E_2 behaves in exactly the same way in both E_1 and E_2 during the first $g/4$ rounds. Task γ is not performed in execution E_2 by round $g/4$, because the stations in B have been failed and the remaining ones behave as in E_1.

Station $\mathcal{P}$ is not failed during E_2 hence it does the same both in E_1 and in E_2. Consider an execution E_3: it is like E_2 till round $g/4$, then all the stations except $\mathcal{P}$ are failed. The definition of E_3 is consistent with the power of the adversary UNBOUNDED. Station $\mathcal{P}$ is operational but halted and task γ is still outstanding in E_3 at round $g/4$. We conclude that the algorithm is not reliable. This contradiction completes the proof of the claim.

Hence there are at least $p - g/4 = \Omega(p)$ stations still operational and non-halted in round $g/4$ in E_1, and they have generated work $\Omega(pg)$ by then. □

Corollary 5.5 *Algorithm* TWOLISTS *is asymptotically optimal, among reliable randomized algorithms solving Do-All in the channel without collision detection, against the adversary f-*BOUNDED*, for $0 \leq f < p$.* □

This shows that randomization does not help against the strongest possible size-bounded adversaries for the channel without collision detection. In Section 5.3 we show that randomization can make a difference for this channel in weaker adversarial models.

5.2 Channel with collision detection

We develop a deterministic algorithm GroupsTogether. The algorithm is a suitable modification of TwoLists. The main difference is that while TwoLists avoids conflicts by its design, the algorithm in this section uses conflicts aggressively as a hedge against faults.

Station $\mathcal{P}_v$ in this algorithm maintains four lists. Lists $\texttt{TASKS}_v$, $\texttt{DONE}_v$ and $\texttt{LOCALTASKS}_v$ are the same as in Section 5.1. The stations are partitioned into groups, which are maintained as a circular list `GROUPS`. There is a pointer which points to `Group`. The stations in the same group always perform the same tasks and broadcast simultaneously. The tasks are assigned to groups as follows: $\texttt{Task}_v$ is assigned to `Group`, then the next task is assigned to the next group, and so on in the circular orders on both lists.

The way the lists are updated after a scheduled broadcast depends on the fact if a broadcast happened, which is detected by receiving either a message or a signal of collision. When a broadcast happens then the tasks performed by the group just heard on the channel are removed from $\texttt{TASKS}_v$. If a broadcast did not happen then `Group` is removed from `GROUPS`. If an item which is pointed at by the associated pointer is removed from a list then the pointer is automatically advanced to the following item. Other than that the list pointers are updated as follows: pointer $\texttt{Task}_v$ is moved by $\lceil\sqrt{\texttt{Shift}_v}\rceil$ positions, and pointer `Group` is advanced by one position.

The stations are partitioned initially into $\min\{\lceil\sqrt{t}\rceil, p\}$ balanced groups, this partitioning is never changed in the course of the algorithm. This partitioning defined as follows: a station is in the group determined by its UID modulo $\min\{\lceil\sqrt{t}\rceil, p\}$. The variable $\texttt{Shift}_v$ is initialized to $\lceil t/2\rceil$. The lists $\texttt{TASKS}_v$ and `GROUPS` are initialized to contain all the tasks and groups, respectively, and are ordered by the identifiers of items.

Algorithm GroupsTogether is in phases repeated in a loop, it is described in Figure 5.3.

A phase of algorithm GroupsTogether is given in Figure 5.4.

Lemma 5.6 *Algorithm* GroupsTogether *is reliable.*

Proof: We show three relevant facts:

1. All the station have the same values of local variables, except possibly for lists `DONE` and `LOCALTASKS`.

Proof by induction: we start with the same setting, and each modification is made according to information heard on the channel.

1) $\texttt{Shift}_v := \lceil t/2 \rceil$
2) $\texttt{Counter}_v := t$
3) $\texttt{DONE}_v := \emptyset$
4) $\texttt{LOCALTASKS}_v := \texttt{TASKS}_v$
5) initialize `Group` to the first element of `GROUPS`
6) initialize $\texttt{Task}_v$ to the first element of $\texttt{TASKS}_v$

while not halted **do execute** GroupsTogetherPhase

Figure 5.3: Algorithm GroupsTogether for station $\mathcal{P}_v$.

2. All the tasks which are not in list `TASKS` at any step of computation have been performed.

This is because a task is removed only if the stations that performed it announced its existence.

3. Each station terminates eventually.

This happens if the local list `TASKS` becomes empty but after at most t phases. It follows from the first fact that all the stations halt at the same moment. If they halt because `Counter` equals zero then all the tasks have been performed during Round 1 of each phase. Otherwise lists `TASKS` are empty and, by the second fact, we have that all the tasks have been done. □

Theorem 5.7 *Algorithm* GroupsTogether *solves Do-All with the minimum work* $\Theta(t + p\sqrt{t})$ *against the adversary* f-Bounded, *for* $0 \le f < p$.

Proof: Epochs are defined as in the proof of Theorem 5.3. If no broadcast is heard at a step then the work performed by all the stations at this step and the stations that were to broadcast (since their last successful broadcast) is at most $2p$. There are at most $\mathcal{O}(\sqrt{t})$ such epochs because of the total number of groups. (This is the main difference with the proof of Theorem 5.3.)

The rest of proof is similar to the Theorem 5.3. We estimate the work performed by broadcasting groups during an epoch. The work performed while $|\texttt{GROUPS}|^2 \le |\texttt{TASKS}|$ is estimated as follows. Each task performed and then reported was performed by only one group in that way. If groups contain $\mathcal{O}(1)$ stations then this can be charged to $\mathcal{O}(t)$, otherwise to $\mathcal{O}(p\sqrt{t})$. Consider next the work performed

ROUND 1:

Step 1: pause

Step 2:

- perform the first task after $\texttt{Task}_v$ not performed yet and remove it from $\texttt{LOCALTASKS}_v$
- decrease $\texttt{Counter}_v$ by 1
- if $\texttt{Counter}_v \leq 0$ then halt

ROUND 2:

Step 1: if v belongs to $\texttt{Group}$ then broadcast one bit

Step 2: attempt to receive a message and do the following:

- if a broadcast is heard, including a collision, then
  a) compute list $\texttt{DONE}_w$ for any w in $\texttt{Group}$ and remove its elements from $\texttt{TASKS}_v$
  b) if $\texttt{TASKS}_v$ is empty then halt
  c) advance the pointer on $\texttt{STATIONS}_v$ by one position

  else remove $\texttt{Group}$ from $\texttt{GROUPS}$
- if $|\texttt{TASKS}_v| < \texttt{Shift}_v$ then $\texttt{Shift}_v := \lceil \texttt{Shift}_v/2 \rceil$
- advance the pointer on $\texttt{TASKS}_v$ by $\lceil \sqrt{\texttt{Shift}_v} \rceil$ positions
- perform the assigned task and add it to $\texttt{DONE}_v$

Figure 5.4: GROUPSTOGETHERPHASE for station $\mathcal{P}_v$.

while $|\texttt{GROUPS}|^2 > |\texttt{TASKS}|$. The number stored in the variable $\lceil\sqrt{\texttt{Shift}}\rceil$ has the property that these many groups in list GROUPS perform distinct tasks in a step. It follows from the similar observation as in proof of Theorem 5.3. Hence each of these groups was heard, if not failed, so this work can be charged to $\mathcal{O}(p\sqrt{\texttt{Shift}})$.

The values of Shift make a geometrically decreasing sequence, hence summing over epochs gives the estimate $\mathcal{O}(p\sqrt{t})$. □

The fact that the algorithm GroupsTogether needs to perform only the minimum amount of work has the following two consequences:

Corollary 5.8 *Algorithm* GroupsTogether *is asymptotically optimal, among reliable randomized algorithms solving Do-All in the channel with collision detection, against the strongest size-bounded adversaries.* □

Corollary 5.9 *Algorithm* GroupsTogether *cannot be beaten in asymptotic work performance by any randomized reliable algorithm in any* weaker *adversarial model in which not all stations can be failed.* □

5.3 Randomized solutions

We have shown that randomization does not help against strongly-adaptive size-bounded adversaries. Corollary 5.9 implies that this is also the case for the channel with collision detection against weaker adversaries. In this section we show that, as far as the channel *without* collision detection is concerned, the power of a size-bounded adversary does matter if we compare the optimal performance of deterministic algorithms versus randomized ones.

We develop a randomized algorithm MixThenWork. It selects a sufficiently long random set of stations, which then run a suitably modified algorithm TwoLists. The algorithm uses the same lists and pointers as TwoLists. Additionally, there is a cyclic list SELECTED of stations, also with a pointer. Algorithm MixThenWork iterates phases in a loop as described in Figure 5.5. The process of random selection of stations is performed by procedure Mixing, given in Figure 5.6.

It remains to describe procedure AltTwoLists. It operates by having the stations on list SELECTED run TwoLists, with SELECTED playing the role of STATIONS. The remaining stations in STATIONS, if any, do not make any attempts to broadcast then. Instead, they listen to the channel to learn about tasks performed by other

while not halted **do**

- **if** $|\texttt{TASKS}_v| < |\texttt{STATIONS}_v|^2$ **then** perform $\sqrt{|\texttt{TASKS}_v|}$ many MIXING phases
  else move all the stations to SELECTED
- **if** SELECTED= $\emptyset$ **then** append the first element of $\texttt{STATIONS}_v$ to SELECTED
- call ALTTWOLISTS

Figure 5.5: Algorithm MIXTHENWORK for station $\mathcal{P}_v$.

Step 1:

- if $\mathcal{P}_v$ is in $\texttt{STATIONS}_v$ the toss a coin with the probability $|\texttt{STATIONS}_v|^{-1}$ of heads to come up
- if heads came up then broadcast v on the channel

Step 2:

- attempt to receive a message
- if number w was heard then remove station $\mathcal{P}_w$ from $\texttt{STATIONS}_v$ and append it to SELECTED
- if $\texttt{STATIONS}_v$ is empty then return control to the algorithm

Figure 5.6: Phase MIXING for station $\mathcal{P}_v$.

stations and keep performing consecutive tasks still in `TASKS`. A task is removed from `TASKS` by a station only if it was performed by a station that managed to broadcast on the channel, otherwise it is just marked as done. This allows to maintain the same items on instances of the list `TASKS` by all the stations. A station halts when it has all the tasks on its copy of `TASKS` marked as done or list `LOCALTASKS` becomes empty. Procedure AltTwoLists returns control, and a new iteration starts, as soon as all the stations in `SELECTED` fail to broadcast in their assigned time slots.

Lemma 5.10 *Algorithm* MixThenWork *is reliable.*

Proof: First observe that each call of AltTwoLists terminates after $\mathcal{O}(t + p^2)$ phases since it simulates algorithm TwoLists and if all the stations from `SELECTED` become faulty then it switches control to the algorithm. Each station halts after $\mathcal{O}(p \cdot (t + p^2))$ because after at most p mixing phases one station is moved from $\texttt{STATIONS}_v$ to `SELECTED`.

Note that list $\texttt{TASKS}_v$ is the same in all the stations, and if any task is not on this list then it has been performed already. Stations can halt only during a call of procedure AltTwoLists, hence if all the stations from `SELECTED` halt during some phase then the others do the same in the same step. When algorithm TwoLists halts then all the tasks have already been performed because all the lists $\texttt{TASKS}_v$ are empty. □

Theorem 5.11 *Algorithm* MixThenWork *solves Do-All in the channel without collision detection with the minimum expected work* $\Theta(t+p\sqrt{t})$ *against any* Weakly-Adaptive Linearly-Bounded *adversary.*

Proof: If $p^2 < t$ then all the stations are in the list `SELECTED` from the start and all stations execute TwoLists algorithm performing work $\mathcal{O}(t)$. Consider the case $p^2 \geq t$. A new station is added to the list `SELECTED` in a phase of Mixing with some constant probability. The number of phases of the first call of Mixing is $\mathcal{O}(\sqrt{t})$. It follows from the definition of the adversarial model and by the Chernoff bound that there are $\Omega(\sqrt{t})$ stations in `SELECTED` that are not failure-prone, after this first call, with the probability at least $1 - e^{-a\sqrt{t}}$, for some $a > 0$. If this event holds then there is just one iteration of the loop in the algorithm, and the bound $\mathcal{O}(p\sqrt{t})$ on the work follows. This is because $\Theta(\sqrt{t})$ stations execute algorithm TwoLists, and a fraction of them may fail. Hence the work done by these $\Theta(\sqrt{t})$ stations is $\mathcal{O}(t)$ by Theorem 5.3. In the meantime some other $\mathcal{O}(p)$ stations listening to the channel perform work that is $\mathcal{O}(p/\sqrt{t})$ times greater than the work performed by the selected stations. Hence the total work is

$$\mathcal{O}(t + (p/\sqrt{t})t) = \mathcal{O}(p\sqrt{t}) \ .$$

Otherwise the work is $\mathcal{O}(pt)$. The total expected work is thus of order

$$p\sqrt{t}(1-e^{-a\sqrt{t}})+pte^{-a\sqrt{t}}=\mathcal{O}(p\sqrt{t}) ,$$

which is within the claimed bound. □

Is there an algorithm that needs to perform only the expected minimum work against such weakly-adaptive adversaries who could fail asymptotically more than a constant fraction of the stations? Corollary 5.13 answers this question in the negative for certain values of p, f and t. We show the following precise lower bound:

Theorem 5.12 *The adversary* Weakly-Adaptive f-Bounded *can force any reliable randomized algorithm solving Do-All in the channel without collision detection to perform the expected work* $\Omega(t+p\sqrt{t}+p\min\{f/(p-f),t\})$.

Proof: Part $\Omega(t+p\sqrt{t})$ follows from Lemma 5.1. We show the remaining one. Let number $g\le f$ be a parameter, we will set its value later in the proof. Consider the following experiment: the algorithm is run for g rounds, and each station which wants to broadcast successfully is failed just before it is to do so. Additionally, at round g, a sufficient number of the remaining stations, say, those with the smallest identifiers, is failed to make the total number of failures by round g equal to exactly g. We define a probabilistic space in which the elementary events are the sets of UIDs of stations corresponding to sets of size g which can be failed in the experiment. Let $\mathcal{F}$ denote the family of all such elementary events. The probability $\mathsf{P}(\omega)$ of $\omega\in\mathcal{F}$ is defined to be equal to the probability of an occurrence of an execution during the experiment, in which exactly the stations with UIDs in ω are failed by round g.

The following equality holds

$$\sum_{|A|=p-f}\ \sum_{\omega\cap A=\emptyset}\mathsf{P}(\omega)=\binom{p-g}{p-f} ,$$

where we sum over subsets $A\subseteq[1..p]$ and elementary events $\omega\in\mathcal{F}$, because each $\mathsf{P}(\omega)$ occurs $\binom{p-g}{|A|}$ times on the left-hand side. There are $\binom{p}{p-f}$ subsets $A\subseteq[1..p]$ with $|A|=p-f$. Hence there is some $C\subseteq[1..p]$, with $|C|=p-f$, such that the probability that the UIDs of stations failed in the experiment are all outside C is at least

$$\begin{aligned}\binom{p-g}{p-f}\Big/\binom{p}{p-f} &= \frac{(p-g)!}{(f-g)!}\cdot\frac{f!}{p!}\\ &= \frac{(f+1-g)(f+2-g)\cdot\ldots\cdot(p-g)}{(f+1)(f+2)\cdot\ldots\cdot p)}\\ &= (1-\frac{g}{f+1})(1-\frac{g}{f+2})\cdot\ldots\cdot(1-\frac{g}{p})\\ &\geq (1-\frac{g}{f+1})^{p-f}\\ &= \left[(1-\frac{g}{f+1})^{(f+1)/g}\right]^{g(p-f)/(f+1)}\\ &\geq 4^{-g(p-f)/(f+1)}\\ &= \Omega(1)\ ,\end{aligned}$$

provided

$$g(p-f)/(f+1) = \mathcal{O}(1)\ ,$$

which is the case if g equals

$$h = \lceil f/(p-f)\rceil \leq f\ .$$

Let the adversary declare exactly the stations not in C as prone to failures. Suppose the algorithm is run for $\min\{h,t\}/4$ rounds and each station not in C which is to broadcast successfully is failed just before it attempts to do so. Such an execution is consistent with the power of the adversary. The event that no message is heard during $\min\{h,t\}/4$ rounds has the probability $\Omega(1)$. Simultaneously, the number of operational stations is $\Omega(p)$. None of these stations may halt by round $\min\{h,t\}/4$, by an argument similar to that used in the proof of Theorem 5.4: the algorithm would prove not to be reliable because this round occurs earlier than $\min\{f,t\}/4$. The expected work in such an execution is thus of order

$$\Omega(1)\cdot p\min\{h/4,t/4\} = \Omega(p\min\{h,t\})\ ,$$

which completes the proof. $\Box$

Corollary 5.13 *If $f = p\cdot(1-o(1/\sqrt{t}))$ then the adversary* Weakly-Adaptive *f-*Bounded *can force any algorithm solving Do-All in the channel without collision detection to perform work $\omega(t+p\sqrt{t})$, if only $t=o(p^2)$.* $\Box$

Chapter 6

Conclusions

We studied solutions to the *Do-All* problem in the context of different models of communication and adversary. We addressed the following issues:

- What is the impact of the model of communication, like availability of atomic communication or collision detection?
- Does randomization help to perform tasks?
- How does the efficiency of solutions depend on various adversarial models?

We showed that all these parameters have an impact.

All the presented algorithms, which are analyzed as run against the LINEARLY-BOUNDED adversary, can be modified to have work and communication $\mathcal{O}(t + p^2)$ against the adversary UNBOUNDED. To this end it is sufficient for each non-halted processor to switch at some point to an alternative algorithm which is efficient against the adversary UNBOUNDED, for example to the one developed in [29], assuming additionally that all the tasks are still outstanding. The moment of switching may be defined as the first time step when the total communication exceeds $t + p^2$.

Most of the previous research on the multiple-access channel has concerned the issues of stability of protocols handling dynamically generated packets. There have been quite few static algorithmic problems considered, in which the whole input is provided to the attached stations in advance [12]. The broadcast channel is ubiquitous in local area networks, and all its algorithmic aspects deserve a study, those concerning static problems in particular. The results of this work attempt to demonstrate that this is the case for the problem *Do-All*.

Open problems

The optimality of solving *Do-All* by randomized algorithms against Weakly-Adaptive Linearly-Bounded in message-passing models is an open problem. We claim that the expected work or communication of any such algorithm is $\Omega(p \log^* p)$ for $t = p$.

Another open question is if there is a difference in power between the adversaries Linearly-Bounded and Unbounded in this model, for both deterministic and randomized algorithms. We claim that they have comparable power.

There is no lower bound known for effort required of algorithms solving *Do-All*. It would be especially interesting to have such a bound in the form of a trade-off between work and communication.

Another open issue regards constructiveness of deterministic algorithms. No such algorithm solving *Do-All* with work and communication $\mathcal{O}(t + p \cdot \text{polylog}(t))$ is known, even if run against Linearly-Bounded adversary.

An issue which is not fully understood yet is what is the impact of atomicity in communication in the message-passing model on the efficiency of *Do-All* solutions.

There is an open problem for the multiple-access channel whether the lower bound given in Theorem 5.12 can be matched by the performance of a randomized algorithm.

Bibliography

[1] M. Ajtai, J. Aspnes, C. Dwork, and O. Waarts, A theory of competitive analysis of distributed algorithms, in *Proc. 33rd IEEE Symp. of Foundations of Computer Science*, 1994, pp. 401–411.

[2] N. Alon, and F.R.K. Chung, Explicit construction of linear sized tolerant networks, *Discrete Mathematics*, 72 (1988) 15–19.

[3] N. Alon, H. Kaplan, M. Krivelevich, D. Malkhi, and J. Stern, Scalable secure storage when half the system is faulty, in *Proc. 27th Int. Colloquium on Automata, Languages and Programming*, 2000, Geneva, Switzerland, Springer LNCS 1853, pp. 577–587.

[4] N. Alon, J.H. Spencer, and P. Erdős, "The Probabilistic Method," John Wiley, 1992.

[5] R.J. Anderson, and H. Woll, Algorithms for the certified write-all problem, *SIAM J. Computing*, 26 (1997) 1277-1283.

[6] J. Aspnes, and W. Hurwood, Spreading rumors rapidly despite an adversary, *J. Algorithms*, 26 (1998) 386–411.

[7] H. Attiya, and J. Welch, "Distributed Computing – Fundamentals, Simulations, and Advanced Topics," McGraw-Hill, 1998.

[8] D. Bertsekas, and R.G. Gallager, "Data Networks," Prentice Hall, 1991.

[9] M. Bridgland, and R. Watro, Fault-tolerant decision making in totally asynchronous distributed systems, *Proc. 6th ACM Symp. on Principles of Distributed Computing*, 1987, pp. 52–63.

[10] J. Buss, P.C. Kanellakis, P. Ragde, and A.A. Shvartsman, Parallel algorithms with processor failures and delays, *J. Algorithms*, 20 (1996) 45–86.

[11] N. Carriero, E. Freeman, D. Gelerntr, and D. Kaminsky, Adaptive parallelism with Piranha, *IEEE Computer*, 1995.

[12] B.S. Chlebus, Randomized communication in radio networks, a chapter in "Handbook on Randomized Computing," P.M. Pardalos, S. Rajasekaran, J.H. Reif, and J.D.P. Rolim, (Eds.), Kluwer Academic Publishers, to appear.

[13] B.S. Chlebus, R. De Prisco, and A.A. Shvartsman, Performing tasks on synchronous restartable message-passing processors, *Distributed Computing*, 14 (2001) 49–64.

[14] B.S. Chlebus, S. Dobrev, D.R. Kowalski, G. Malewicz, A.A. Shvartsman, and I. Vřto, Towards practical deterministic write-all algorithms, in *Proc., 13th ACM Symp. on Parallel Algorithms and Architectures*, 2001, Crete, Greece, to appear.

[15] B.S. Chlebus, A. Gambin, and P. Indyk, PRAM computations resilient to memory faults, in *Proc. 2nd European Symposium on Algorithms*, 1994, Utrecht, The Netherlands, Springer LNCS 855, pp. 401–412.

[16] B.S. Chlebus, L. Gąsieniec, D.R. Kowalski, and A.A. Shvartsman, Performing tasks in networks with a constant fraction of non-faulty processors, submitted.

[17] B.S. Chlebus, and D.R. Kowalski, Randomization helps to perform tasks on processors prone to failures, in *Proc. 13th Int. Symposium on Distributed Computing*, 1999, Springer LNCS 1693, pp. 284–296.

[18] B.S. Chlebus, D.R. Kowalski, and A. Lingas, The Do-All problem in broadcast networks, in *Proc. 20th ACM Symp. on Principles of Distributed Computing*, 2001, Newport, Rhode Island, to appear.

[19] H. Davenport, "Multiplicative Number Theory," 2nd ed. revised by H.L. Montgomery, Springer Verlag, Berlin, 1980.

[20] R. De Prisco, A. Mayer, and M. Yung, Time-optimal message-efficient work performance in the presence of faults, in *Proc. 13th ACM Symp. on Principles of Distributed Computing*, 1994, pp. 161–172.

[21] C. Dwork, J. Halpern, and O. Waarts, Performing work efficiently in the presence of faults, *SIAM J. on Computing*, 27 (1998) 1457–1491.

[22] D. Dwork, D. Peleg, N. Pippenger, and E. Upfal, Fault tolerance in networks of bounded degree, *SIAM J. on Computing*, 17 (1988) 975–988.

[23] K. Diks, and A. Pelc, Optimal adaptive broadcasting with a bounded fraction of faulty nodes, *Algorithmica*, 28 (2000) 37–50.

[24] M. Feeley, B. Bershad, J. Chase, and H. Levy, Dynamic node reconfiguration in a parallel–distributed environment, *Proc. 3d ACM Princ. and Practice of Parallel Prog.*, 1991, pp. 114–121.

[25] F. Fich, and E. Ruppert, Lower bounds in distributed computing, *Proc. of the 14th International Symposium on Distributed Computing*, 2000.

[26] R.G. Gallager, A perspective on multiaccess channels, *IEEE Trans. on Information Theory*, 31 (1985) 124–142.

[27] J.F. Groote, W.H. Hesselink, S. Mauw, and R. Vermeulen, An algorithm for the asynchronous write-all problem based on process collision, *Distributed Computing*, 2001, to appear.

[28] L.A. Goldberg, P. MacKenzie, M. Paterson, and A. Srinivasan, Contention resolution with constant expected delay, *J. ACM*, 47 (2000) 1048–1096.

[29] Z. Galil, A. Mayer, and M. Yung, Resolving message complexity of byzantine agreement and beyond, in *Proc. 36th IEEE Symp. on Foundations of Computer Science*, 1995, pp. 724–733.

[30] L. Gąsieniec, A. Pelc, and D. Peleg, The wakeup problem in synchronous broadcast system, in *Proc. 19th ACM Symp. on Principles of Distributed Computing*, 2000, Portland, Oregon, pp. 113–122.

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

[32] A.G. Greenberg, and S. Winograd, A lower bound on the time needed in the worst case to resolve conflicts deterministically in multiple access channels, *J. ACM*, 32 (1985) 589–596.

[33] J. Håstad, T. Leighton, and B. Rogoff, Analysis of backoff protocols for multiple access channels, *SIAM J. on Computing*, 25 (1996) 740–774.

[34] V. Hadzilacos and S. Toueg, Fault-tolerant broadcasts and related problems, in "Distributed Systems," 2nd Ed., S. Mullender, Ed., Addison-Wesley and ACM Press, 1993.

[35] D.L. Kaminsky, Adaptive Parallelism with Piranha, *Doctoral Dissertation*, Yale University, 1994.

[36] J. Komlós, and A.G. Greenberg, An asymptotically nonadaptive algorithm for conflict resolution in multiple-access channels, *IEEE Trans. on Information Theory*, 31 (1985) 303–306.

[37] P.C. Kanellakis and A.A. Shvartsman, Efficient parallel algorithms can be made robust, *Distributed Computing*, 5 (1992) 201–217.

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

[39] Z.M. Kedem, K.V. Palem, M.O. Rabin, and A. Raghunathan, Efficient program transformations for resilient parallel computation via randomization, in *Proc. 24th ACM Symp. on Theory of Computing*, 1992, pp. 306–318.

[40] Z.M. Kedem, K.V. Palem, A. Raghunathan, and P. Spirakis, Combining tentative and definite executions for dependable parallel computing, in *Proc. 23rd ACM Symp. on Theory of Computing*, 1991, pp. 381–390.

[41] Z.M. Kedem, K.V. Palem, and P. Spirakis, Efficient robust parallel computations, in *Proc. 22nd ACM Symp. on Theory of Computing*, 1990, pp. 138–148.

[42] S.C. Kontogiannis, G.E. Pantziou, P.G. Spirakis, and M. Yung, Robust parallel computations through randomization, *Theory of Computing Systems*, 33 (2000) 427–464.

[43] E. Kushilevitz, and Y. Mansour, An $\Omega(D \log(N/D))$ lower bound for broadcast in radio networks, *SIAM J. on Computing*, 27 (1998) 702–712.

[44] A. Lubotzky, R. Phillips and P. Sarnak, Ramanujan graphs, *Combinatorica*, 8 (1988) 261–277.

[45] M. Litzkow, M. Livny, and M. Mutka, Condor – a hunter of idle workstations, *Proc. 8th Int. Symp. on Distributed Computing Systems*, 1988.

[46] N.A. Lynch, "Distributed Algorithms," Morgan Kaufmann, 1996.

[47] G.A. Margulis, Explicit group-theoretical constructions of combinatorial schemes and their applications to the design of expanders and concentrators, *Problems of Information Transmission*, 24 (1988) 39–46.

[48] C.U. Martel, Maximum finding on a multiple access broadcast network, *Information Processing Letters*, 52 (1994) 7–13.

[49] C. Martel, A. Park, and R. Subramonian, Work-optimal asynchronous algorithms for shared memory parallel computers, *SIAM J. Comput.*, 21 (1992) 1070–1099.

[50] C. Martel, and R. Subramonian, On the complexity of certified Write-All algorithms, *J. Algorithms*, 16 (1994) 361–387.

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

[52] P. Raghavan, and E. Upfal, Stochastic contention resolution with short delays, *SIAM J. on Computing*, 28 (1998) 709–719.

[53] M. Saks, N. Shavit, and H. Woll, Optimal time randomized consensus - making resilient algorithms fast in practice, in *Proc. 2nd SIAM–ACM Symp. Discrete Algorithms*, 1991, San Francisco, California, pp. 351–362.

[54] A.S. Tanenbaum, "Computer Networks," Prentice Hall, 1996.

[55] G. Tel, "Introduction to Distributed Algorithms," Cambridge University Press, 2000.

[56] M. Theimer, K. Lantz, and D. Cheriton, Preemptable remote execution facilities for the V-system, *Proc. 10th ACM Symp. on Operating Systems Principles*, 1985, pp. 2–12.

[57] E. Upfal, Tolerating a linear number of faults in networks of bounded degree, *Information and Computation*, 115 (1994) 312–320.

[58] D.E. Willard, Log-logarithmic selection resolution protocols in a multiple access channel, *SIAM J. on Computing*, 15 (1986) 468–477.