ROBUST DISTRIBUTED COOPERATION IN THE PRESENCE OF QUANTIFIED ADVERSITY

Chryssis Georgiou, Ph.D.

University of Connecticut, 2003

The ability to cooperatively perform a collection of tasks in a distributed setting is key to solving a broad range of computation problems ranging from distributed search to distributed simulation and multi-agent collaboration. *Do-All*, an abstraction of such cooperative activity, is the problem of using p processors to cooperatively perform n independent and idempotent tasks in the presence of adversity. The *Do-All* problem can be used to identifying the trade-offs between efficiency and fault-tolerance in distributed cooperative computing. Solutions for *Do-All* may yield insight leading to efficient and fault-tolerant algorithms for distributed co-operation. Although significant research was dedicated to studying *Do-All*, prior work offers only a partial understanding of this problem. In particular, while prior work shows how to achieve fault-tolerance in the presence of adversity, it does not adequately teach how the adverse environment affects the efficiency of *Do-All* solutions. This thesis substantially increases this understanding. One of the contributions includes failure sensitive upper and lower bounds for *Do-All* in certain models of computation, that show how failures affect the efficiency of *Do-All* solutions. The upper/lower bounds are given as functions of n, p *and* f, the number of failures caused by the adverse environment. Another contribution of the thesis is the definition and analysis of the *iterative Do-All* problem, that models the repetitive use of *Do-All* algorithms, such as found in typical algorithm simulations.

This thesis also studies the distributed cooperation problem in partitionable networks, where partitions may interfere with the progress of the computation. Group communication services are used to develop robust algorithms for this settings. Moreover, it is shown that it is possible to obtain optimally-competitive scheduling algorithms in partitionable networks by proving upper and lower bound results. These results demonstrate precisely how partitions affect the efficiency of computation.

Overall, the thesis is substantially contributing to the study of the trade-offs between efficiency and fault-tolerance in cooperative computing and is advancing the state-of-the-art in principles of robust distributed computing.

ROBUST DISTRIBUTED COOPERATION

IN THE PRESENCE OF QUANTIFIED ADVERSITY

Chryssis Georgiou

M.S., Computer Science & Engineering, University of Connecticut, 2002
B.S., Mathematics, University of Cyprus, 1998

A Dissertation

Submitted in Partial Fulfillment of the

Requirements for the Degree of

Doctor of Philosophy

at the

University of Connecticut

2003

APPROVAL PAGE

Doctor of Philosophy Dissertation

ROBUST DISTRIBUTED COOPERATION

IN THE PRESENCE OF QUANTIFIED ADVERSITY

Presented by

Chryssis Georgiou, M.S., B.S.

Major Advisor ______________________________
Alex A. Shvartsman

Associate Advisor ______________________________
Nancy A. Lynch

Associate Advisor ______________________________
Alexander Russell

University of Connecticut

2003

ACKNOWLEDGEMENTS

This dissertation would have been impossible without the encouragement, guidance and support of the three members of my committee.

I am deeply indebted to my advisor and mentor, Alex Shvartsman, for giving me the opportunity to pursue doctoral studies at the University of Connecticut. His continuous support and advice brought these studies to a successful completion. I thank him for his critical and pertinent criticism on my work that helped me grow both as a researcher and as a person. His incessant desire for excellence has positively influenced the presentation and organization of this dissertation. I thank him for the time he invested on me and the financial support he provided me over the years of my graduate studies.

My gratitude to Alexander Russell is indefinable. He has played a significant role in my graduate education and my growth as a researcher in the field of theoretical computer science. My interaction and collaboration with him helped me to advance my knowledge and understanding on how to apply mathematical reasoning in computer science. I thank him for the time he dedicated to me and the mathematical tools he enriched me with.

I am particularly grateful to Nancy Lynch, first, for agreeing to be on my advisory committee, and second, for giving me the opportunity to present parts of my work at the MIT TDS seminar. Her feedback along with the comments of the other members of her group at MIT have significantly improved the quality of my work.

During my graduate studies at UConn I had the opportunity to collaborate on research projects with Dariusz Kowalski, Antonio Fernández and Peter Musial. I am thankful for the knowledge and experience I obtained through these wonderful collaborations. I would also like to express my gratitude to Lester Lipsky, Thomas Peters, and Eugene Santos Jr. for valuable discussions on course-related projects. My thanks to Dina Goldin and Kishori Konwar for our collaboration in teaching the Algorithms and Complexity class in the Fall of 2001. I am also thankful to all the officemates I had at UConn during my studies, especially to Cecilia Bastarrica, Peter Musial, and Mariam Momenzadeh.

I am obliged to Marios Mavronicolas for his guidance and advice during my undergraduate studies at the University of Cyprus and for encouraging me to pursue doctoral studies in theoretical computer science.

Finally, I would like to thank my family. My parents, Georgios and Michaela Georgiou, for sacrificing their personal lives for me. They have been supporting and encouraging me to realize my dreams since the day they brought me to this world. My brother, Demetris Georgiou, for always believing in me and supporting all my decisions. My fiancé and soon my-wife-to-be, Agni Stylianou, for everything. There is no way I can measure the many ways she has unconditionally supported me over the years it took me to complete my studies. She has been my strength and joy. I dedicate this dissertation to all four members of my family.

CREDITS

This dissertation incorporates research results appearing in the following publications:

[48, 49] This is a joint work with A. Russell and A. Shvartsman. [48] will appear in *Distributed Computing*. A preliminary version [49] appears in the Proceedings of *DISC'01*. This work corresponds to Sections 4.1-4.3, 5.1, 6.1 and parts of Sections 3.1-3.4 of the dissertation.

[47] This paper is a joint work with D. Kowalski and A. Shvartsman. It appears in the Proceedings of *DISC'03*. It corresponds to Section 5.2.

[51] This paper is a joint work with A. Russell and A. Shvartsman. It appears in the Proceedings of *OPODIS'02*. It corresponds to Section 6.2 and parts of Section 3.4.

[53, 54] This is a joint work with A. Shvartsman. [53] appears in the *Journal of Discrete Algorithms*, 2003. A preliminary version [54] appears in the Proceedings of *SIROCCO'00*. This work corresponds to Section 7.1 and parts of Sections 3.1-3.4.

[52] This paper is a joint work with A. Russell and A. Shvartsman. It appears in the Proceedings of *STOC'03*. It corresponds to Section 7.2 and parts of Sections 3.2-3.3.

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1

Introduction

This thesis studies the impact of the adverse environment on the efficiency of distributed cooperative computing.

1.1 Motivation for this research

The ability to cooperatively perform a collection of tasks in a distributed setting is key to solving a broad range of computation problems ranging from distributed search (e.g., SETI [74]) to distributed simulation (e.g., [25]) and multi-agent collaboration (e.g., [2, 108]). Therefore, cooperative computing has drawn a lot of attention from the research community in the last two decades and substantial research was dedicated to investigating how processors can cooperate effectively in order to exploit parallelism in a system consisting of multiple processing elements.

Distributed systems consisting of hundreds and thousands of processing units (e.g., multiprocessor machines, clusters of workstations, wide-area networks) are widely used. In such systems it is possible that the set of processing elements available to the computation and their ability to communicate may dynamically change due to perturbations in the computation medium. Such changes may degrade the efficiency of algorithms designed to solve computational problems on these multiprocessing systems, and cause algorithms to produce incorrect results.

Therefore, there is a corresponding need for the development of efficient and dependable algorithms that are able to cope with unpredictable changes in the computation medium caused by component failures or delays. Algorithms need to be both efficient and fault-tolerant. We call such algorithms *robust*. However, developing robust algorithms for distributed cooperation is inherently difficult since *fault-tolerance* is achieved by *introducing redundancy*, while *efficiency* is achieved by *eliminating redundancy*.

To study aspects of the trade-off between efficiency and fault-tolerance in cooperative computing and to obtain insight into developing robust algorithms for distributed cooperation, past research (e.g., [33, 68, 17, 44, 28]) has focused on studying the abstract problem of performing a set of tasks in a decentralized setting, known as the *Do-All* problem.

Do-All: *p processors must cooperatively perform n tasks in the presence of adversity.*

In the *Do-All* problem, the tasks are assumed to be similar, independent and idempotent. By the similarity of the tasks we mean that the task executions consume equal or comparable resources. By the independence of the tasks we mean that the completion of any task does not affect any other task. By the idempotence of the tasks we mean that each task can be executed multiple times or concurrently and produce the same final result.

Several high-level computational problems can be abstracted in terms of the *Do-All* problem. For example, in image processing [112] and computer graphics [42], a significant amount of data processing (e.g., operations on large data structures, computing complicated partial and ordinary differential equations) is required, especially in visualization (achieving graphical visual realism of real world objects) [89, 101]. When the data to be computed can be decomposed into smaller independent "chunks", a usual approach is to load-balance the chunks among the different processing units of a parallel machine (or a cluster of machines) [101, 58]. The data chunks can be abstracted as *Do-All* tasks and the processing units can be abstracted as *Do-All* processors. In databases [36], when querying a large (unsorted) data space, it is often desirable to use multiple machines to search different records of the database in an attempt to decrease the search time [1]. In fluid dynamics, researchers study the behavior of fluids in different settings by running simulations that involve solving numerically complicated differential equations over large data spaces. Again, when the data can be decomposed into smaller independent chunks, the chunks are assigned on different multiprocessing units to achieve faster and reliable computation [55, 65]. Another example can be found in Cryptography. In particular, in breaking cryptographic schemes. The goal is to search and find a user's private key. A key may be a string of 128 bits, meaning that there are 2^{128} different strings that a user could choose as his private key. Among the various techniques available, the most frequently used is exhaustive search where multiple processing units search simultaneously for the key, each unit searching different sets of bit permutations [106]. Each set of bit permutation can be abstracted as a *Do-All* task and each processing unit can be abstracted as a *Do-All* processor. In general, any problem that involves performing a number of similar independent calculations can be abstracted in terms of the *Do-All* problem.

As we will see in Section 1.3 and more extensively in Chapter 2, prior research offers only a partial understanding of the *Do-All* problem. Specifically, there is a partial understanding on how the adverse environment (e.g., failures) affects the efficiency of *Do-All* solutions, and more generally, how it affects cooperative computing — quantification of adversity does not figure prominently in complexity results. This is rather surprising, especially since research conducted for other fundamental problems of distributed computing, for example the *consensus* problem (a set of processors must agree on a common value, see Section 2.7), always focused on how adversity affects the efficiency and doability of the problem.

The underlying theme that we address in this thesis is

Understanding precisely how the adverse environment affects
the efficiency of cooperative computing.

1.2 Background

The *Do-All* problem and variations of this problem have been studied in a variety of settings, e.g., in *shared-memory* models [68, 86, 59, 7], in *message-passing* models [33, 28, 22, 44] and in *partitionable networks* [32, 83].

In message-passing models, processors communicate by exchanging messages while in shared-memory models processors communicate by reading from and writing to shared memory. In shared-memory models, the *Do-All* problem is known as the *Write-All* problem — *given a zero-valued array of* n *elements and* p *processors, write value* 1 *into each array location* — and it was introduced by Kanellakis and Shvartsman [67]. The main difference between the *Do-All* problem in message-passing models and the *Write-All* problem in shared-memory models is that in *Do-All* the tasks may be supplied to the processors from some external source, while in *Write-All* the tasks are stored in shared-memory accessible to all processors. In particular, each location of the *Write-All* input array may be associated with a task, and when a processor writes the value 1 into a specific location of the input array, this implies that the processor has performed the associated task. In the context of this thesis we abstract away from the source and the nature of the tasks and we treat *Do-All* and *Write-All* as the same problem. However, when we study *Do-All* in shared-memory models, we refer explicitly to *Write-All*.

Do-All has also been studied in the setting of processor groups in partitionable networks, i.e., when the topology of the network may dynamically change due to changes in the communication medium [32, 83]. In this setting, the goal is to utilize the resources of every component of the system during the entire computation. We call this problem *Omni-Do* — *a set of n tasks must be performed by p processors in a distributed system, where each processor must learn all results* — and it was introduced by Dolev, Segala and Shvartsman [32].

Do-All has been also studied under the assumption of *perfect knowledge* [68], where message-passing and shared-memory issues are abstracted away by the assumption of an *oracle* that performs the load-balancing computation on behalf of the processors.

Finally, *Write-All* algorithms have been used in developing simulations of failure-free algorithms on failure-prone processors, e.g., [72, 104, 85, 68]. This is done by iteratively using a *Write-All* algorithm to simulate the steps of failure-free processors on failure-prone processors. In this thesis we abstract this iterative use of *Do-All* algorithms as the r-*iterative Do-All* problem — *using p processors, solve r instances of n-task* Do-All *with the added restriction that every task of the ith instance must be completed before any task of the $(i+1)$st instance is begun*. (The r-*iterative Write-All* and r-*iterative Omni-Do* problems are defined similarly.)

The efficiency of algorithmic solutions to *Do-All* is usually assessed in terms of *work*, *time* and *communication* complexity, depending on the specific model of computation. Work is defined either as the total number of computational steps taken by all available processors during the computation (known as *available processor steps*, introduced by Kanellakis and

Shvartsman [67]) or as the total number of only *task-oriented* computational steps taken by the processors (introduced by Dwork, Halpern, and Waarts [33]). A computational step taken by a processor is said to be task-oriented, if during that step the processor performs a *Do-All* task. We refer to the first variation of work as *work* and we denoting it by S. We refer to the second variation of work as *task-oriented work* and we denote it by W. In synchronous systems, time is defined as the total number of parallel steps required for the computation to terminate. In asynchronous systems, time is defined as the total number of time-units required for the computation to terminate, where a time-unit is usually defined in terms of the clock-ticks of a global clock (that may or may not be accessible to processors). Communication complexity or *message complexity* is defined as the total number of point-to-point messages sent during the computation. We denote it by M.

A trivial lower bound on work for *Do-All* is $\Omega(n)$, since each task has to be performed at least once. A trivial solution to *Do-All* can be obtained by having each processor obliviously perform each of the n tasks. This solution has work $\Theta(n \cdot p)$ and requires no communication. To this respect, a *Do-All* algorithm is considered *efficient* if it achieves work substantially better than the oblivious algorithm. In particular, we say that a *Do-All* algorithm is *optimal* if it can achieve work $O(n)$, *polylogarithmically efficient* if it can achieve work $O(n \log^{O(1)} n)$ and *polynomially efficient* if it can achieve work $O(n^{1+\varepsilon})$, for some $\varepsilon \in (0, 1)$.

1.3 Prior Work

In this section we overview related research. An extensive literature review is given in Chapter 2.

Synchronous message-passing algorithms solving *Do-All* with processor *crashes* (a faulty processor stops all activities and does not perform any further actions) have been provided by Dwork, Halpern and Waarts [33], by De Prisco, Mayer and Yung [28], and by Galil, Mayer and Yung [44]. (The analysis in [33] uses the *task-oriented work* measure that allows processors to idle whereas the analyses in [28] and [44] use the *work* measure where idling processors are charged.) These algorithms use point-to-point messaging and tolerate up to $p-1$ processor crashes. The algorithm by Galil *et al.* [44] (the best among these algorithms) has work $S = O(n + fp)$ and message complexity $M = O(fp^{\varepsilon} + p\min\{f+1, \log p\})$, where f is number of crashes $(f < p)$ and $0 < \varepsilon < 1$. These deterministic algorithms rely on single coordinators or checkpointing strategies for sharing the knowledge about the progress of a computation. Such strategies are subject to the lower bound of $\Omega(n + (f+1)p)$ on work [28]. Chlebus, De Prisco and Shvartsman [17] developed an algorithm – Algorithm AN – that beats this lower bound by using a strategy involving multiple coordinators. It has work $S = O(\log f(n+p\log p/\log\log p))$ and message complexity $M = O(n+p\log p/\log\log p+pf)$, $f < p$. However algorithm AN uses *reliable multicast* [60], which is a strong assumption: if a processor crashes while multicasting a message, then either all non-faulty processors deliver the message or none do. Some local area networks (LANs) might approximate this assumption, but in general it is too costly (or impossible) to provide in many types of distributed systems. Chlebus, Gasieniec, Kowalski and Shvartsman [19] pursued an approach that uses point-to-point messaging and avoids the use of coordinators and checkpointing and developed an algorithm with the combined work and message complexity of $O(n + p^{1.77})$, for all $f < p$. Observe that the work bound is close to the quadratic bound obtained by the oblivious algorithm (where each processor performs all tasks). All of the above give rise to the following question regarding synchronous message-passing *Do-All* algorithms with processor crashes: "Can we develop algorithms that obtain better work and message complexity than the existing ones and that use only point-to-point messaging?". This thesis gives a positive answer to this question.

Chlebus, De Prisco and Shvartsman [17] developed a message-passing solution for *Do-All* – Algorithm AR – that can tolerate processor crashes and *restarts* (a faulty processor may resume computation). Like algorithm AN, algorithm AR uses reliable multicast. It remains an open problem whether it is possible to develop efficient message-passing algorithms that solve *Do-All* for processor crashes and restarts, without the assumption of reliable multicast. It is also worth mentioning that prior work did not consider the *iterative Do-All* problem in message-passing systems. We define and study *iterative Do-All* in this thesis.

In shared-memory models, *Write-All* has been studied in synchronous systems under processor crashes (e.g., [67, 66, 68]), in synchronous systems under processor crashes and restarts (e.g., [15, 68]) and in asynchronous systems (e.g., [85, 87, 59, 68, 7, 18]). Also, Kanellakis, Michailidis and Shvartsman [66], considered the *Write-All* problem for crash-prone processors in a synchronous shared-memory model where the *memory access concurrency* needs to be controlled. The write (resp. read) concurrency is measured as the "redundant" write (resp. read) memory accesses: consider a step of a parallel computation where x processors concurrently write to the same memory location the same value. Then these writes are redundant, since a single write should suffice. Hence, the write concurrency for this step is $x - 1$. Read concurrency is measured in a similar manner.

Write-All algorithms can be used iteratively to simulate parallel algorithms formulated for synchronous failure-free processors on failure-prone processors (e.g., [72, 104, 68]). It was shown that the execution of a single n-processor step on p failure-prone processors does not exceed the work complexity of solving a n-size instance of *Write-All* using p failure-prone processors. By iteratively using algorithm W ([67]), Kanellakis and Shvartsman [68] gave the first upper bound for *iterative Write-All* under processor crashes. In a similar manner, by iteratively using algorithm KMS ([66]), Kanellakis, Michailidis and Shvartsman [66] gave the first upper bound for *iterative Write-All* under processor crashes in the shared-memory model where memory access concurrency needs to be controlled. We note that the bounds [68, 66] on *iterative Write-All* do not adequately demonstrate how the work complexity depends on the number of failures f.

Prior lower/upper bound results for *Do-All* in message-passing and shared-memory models do not teach adequately how the work complexity depends on the number of failures. That is, work was typically given as a function of n and p, but it was either not elucidated how f impacts work, or, when f was a part of the equation, it was primarily due to the nature of a specific algorithm, and not due to the inherent properties of the *Do-All* problem. For example, the work of the best known synchronous shared-memory algorithm (algorithm W) is given as a function of n and p ($S = O(n + p \log n \log p / \log\log p)$) [67]. The work of the best synchronous shared-memory algorithm with controlled memory access concurrency (algorithm KMS) is also given only as a function of n and p ($S = O(n + p \log^2 n \log^2 p / \log\log n)$) [66]. This is also the case with the best known asynchronous shared-memory algorithm (algorithm AW$^{\mathrm{T}}$, $S = O(np^{\varepsilon}),\ \forall\varepsilon > 0$) [7]. Similarly, the best known lower bound for shared-memory models ($S = \Omega(n + p \log p)$) [71] and the best known lower bound applicable to message-passing models ($S = \Omega(n + p \log p / \log\log p)$) [15] do not involve f. The work of message-passing algorithms, e.g., [28, 44], typically *does* include f, but this is due to the use of single coordinators (see discussion above), which means that for f coordinator failures the work necessarily includes an additive term $f \cdot p$. Two message-passing algorithms (algorithms AN and AR) use multiple coordinators [17] to avoid this inefficiency and include a term in the bound on work that depends on $\log f$, but this term is due to the use of multiple coordinators (hence it is due to the nature of the specific algorithms) and not due to the inherent properties of the *Do-All* problem. Obtaining *failure-sensitive* lower/upper bounds for *Do-All* that demonstrate precisely how failures affect *Do-All* efficiency, is important in identifying the trade-offs between efficiency and fault-tolerance in cooperative computing. As mentioned before, this is the main focus of this thesis.

In partitionable networks, the first solution for *Omni-Do* given by Dolev, Segala and Shvartsman [32], considers the case of group *fragmentations*: changes in the communication medium may partition (fragment) the network into several connected components, called *groups*. No group merges are considered. They developed a work-efficient algorithm, called AF, that uses a group communication service [95] to provide membership and communication services to the processors. Algorithm AF has work $S = O(n+n \cdot f)$, where f is the total number of new groups created due to fragmentations minus one (for example, if a group fragments into k new groups, $f = k - 1$). We note that in [32] the message complexity of algorithm AF was not analyzed since obtaining message efficiency was not one of the goals in that paper. However, given the full details of the algorithm it is not difficult to observe that the message complexity of AF is at least quadratic. For the case of fully dynamic changes (including fragmentations and merges), in the same paper, Dolev *et al.* showed that the termination time of any on-line *Omni-Do* algorithm is greater than the termination time of an off-line *Omni-Do* algorithm by a factor greater than $p/12$. They also developed an efficient scheduling strategy for minimizing the execution redundancy showing that it is possible to schedule $\Theta(n^{\frac{1}{3}})$ tasks with at most one common task for any two processors.

Malewicz, Russell and Shvartsman [83, 84] extended the scheduling strategy of Dolev *et al.* [32]. They introduced the notion of k-*waste* that measures the worst-case redundant work performed by k groups (or processors) when started in isolation and merged into a single group at some later time. They adequately investigate the case of 2-*waste* and they show that the work redundancy increases gracefully as the number of tasks performed in isolation increases.

Thus prior work regarding the *Omni-Do* problem established reasonably tight (in the length of the processor schedule) results for a *single* merge, illustrated the fact that on-line algorithms subject to diverging reconfiguration patterns incur linear (in p) overhead relative to an off-line algorithm, and showed an upper bound for an algorithm using group communication services for a limited pattern of network reconfigurations (fragmentations). In this thesis we substantially increase the understanding of solving *Omni-Do* and we demonstrate precisely how the changes in the network topology affect the efficiency of *Omni-Do* algorithms.

1.4 Summary of Contributions

This dissertation substantially advances the understanding on how the adverse environment affects the efficiency of distributed cooperative computations. One of the contributions includes upper and lower bounds for *Do-All* in certain models of computation, that are given not only as a function of the number of tasks n and the number of participating processors p, but also as a function of the number of failures f caused by the adverse environment during the computation. Another contribution of the thesis is the definition and analysis of the *iterative Do-All* problem, that models the repetitive use of *Do-All* algorithms. This thesis also studies the distributed cooperation problem in partitionable networks, where partitions may interfere with the progress of the computation. Group communication services are used to develop robust algorithms for this settings. Moreover, it is shown that it is possible to obtain optimally-competitive scheduling algorithms in partitionable networks by proving upper and lower bound results. These results demonstrate precisely how partitions affect the efficiency of computation. Overall, the dissertation is substantially contributing to the study of the trade-offs between efficiency and fault-tolerance in cooperative computing and is advancing the state-of-the-art in principles of robust distributed computing.

We now overview the technical accomplishments detailed in later chapters of the thesis. The thesis presents *Do-All* lower bounds on work for synchronous crash-prone processors that capture the dependence of work not only on n and p, but also on f, the number of crashes, for the enire range of f ($1 \leq f < p$). Specifically we show that work $S = \Omega(n + p\log p/\log(p/f))$[1] is required to solve *Do-All* when $f \leq p/\log p$, and work $S = \Omega(n + p\log p/\log\log p)$ is required when $f > p/\log n$. This gives the first non-trivial lower bound on *Do-All* work for a moderate number of crashes ($f \leq p/\log p$). For the model of computation where processors are able to make perfect load-balancing decisions locally (the perfect knowledge assumption), matching upper bounds are given. Another contribution of the thesis is the definition and analysis of the r-*iterative Do-All* problem that models the repetitive use of *Do-All* algorithms such as found in algorithm simulations. Our failure-sensitive analysis enables us to derive tight bounds for r-*iterative Do-All* work, that are stronger than the r-fold work complexity of a single *Do-All*. Our approach that models perfect load-balancing allows for the analysis of specific algorithms to be divided into two parts: (i) the analysis of the cost

[1] It is understood that when $f = 0$, then $x/\log(y/f) = 0$, for any $x \neq 0$ and $y \neq 0$.

of tolerating failures while assuming "free" load-balancing, and (ii) the analysis of the cost of implementing load-balancing.

We demonstrate the utility and generality of the above approach by improving the analysis of three known efficient algorithms: (a) We derive a new and complete failure sensitivity analysis of the best known algorithm for the synchronous shared-memory model (algorithm W [67]). Specifically we show that algorithm W solves the *Write-All* problem under processors crashes with work $S = O(n + \log n \log p / \log(p/f))$ when $f \leq p \log p$, and work $S = O(n + \log n \log p / \log \log p)$ when $f > p \log p$, f being the number of crashes. (b) We improve the analysis of the work and message complexity for an efficient synchronous message-passing algorithm (algorithm AN [17]). We show that algorithm AN solves the *Do-All* problem under processor crashes with work $S = O(\log f(n + p \log p / \log(p/f)))$ and message cost $M = O(n + p \log p / \log(p/f) + pf)$ when $f \leq p/\log p$ and, $S = O(\log f(n + p \log p / \log \log p))$ and $M = O(n + p \log p / \log \log p + pf)$ when $f > p/\log p$. (c) We derive a new and complete failure sensitivity analysis on the work of the best known algorithm for the synchronous shared-memory model where the memory access concurrency needs to be controlled (algorithm KMS [66]). Specifically, we show that algorithm KMS achieves work $S = O(n + p \log^2 n \log^2 p / \log(p/f))$ when $f \leq p/\log p$, and work $S = O(n + p \log^2 n \log^2 p / \log \log p)$ when $f > p/\log p$. For each of the three algorithms, substantial improvement in the analysis is recorded, especially for a moderate number of failures ($f \leq p/\log p$). Finally, by iteratively using algorithms W, KMS, and AN and using our new approach to their failure-sensitive analyses, we obtain tighter upper bounds for the *iterative Write-All* problem in shared-memory systems, and the first non-trivial upper bound analysis of the *iterative Do-All* problem in message-passing systems.

Another contribution of the thesis is the development of a new robust algorithm for p synchronous processors that solves the *Do-All* problem with n tasks in the presence of up to f crashes ($f < p$) with work complexity $S = O(n + p\min\{f+1, \log^3 p\})$ and message complexity $M = O(fp^\varepsilon + p\min\{f+1, \log p\})$, for any $\varepsilon > 0$. This result improves the work complexity $S = O(n + fp)$ of the algorithm of Galil *et al.* [44] mentioned in the previous section, while obtaining the same message complexity. It also improves on the algorithm of Chlebus *et al.* [19], also mentioned in the previous section, that has $S = O(n + p^{1.77})$ and $M = O(p^{1.77})$. Unlike algorithm AN [17] that has comparable work complexity (even using our new failure-sensitive analysis) but uses reliable multicast, the new algorithm uses simple point-to-point messaging. The algorithm uses an approach for sharing knowledge among processors that is less authoritarian than the use of coordinators and checkpointing (as used in previously developed algorithms in the same setting). Instead, it uses an approach where processors share information using a new gossip algorithm, where the point-to-point messaging is constrained by means of a communication graph that represents a certain subset of edges in a complete communication network. The processors decide where to send a gossip message based on sets of permutations with special combinatorial properties that we show to exist. This gossip algorithm tolerates up to $p-1$ processor crashes and it runs in $O(\log^2 p)$ time and sends $O(p^{1+\varepsilon})$ messages, for any $\varepsilon > 0$. This result substantially improves on the message complexity $M = O(p^{1.77})$ of the previously best known gossip algorithm of Chlebus and Kowalski [21], while obtaining the same asymptotic time complexity.

The thesis also substantially contributes to the study of the *Omni-Do* problem in partitionable networks, where algorithms must deal with groups of processors that become disconnected and reconnected during the computation. We present a robust algorithm, called algorithm AX, that solves the *Omni-Do* problem for asynchronous processors under group fragmentations and merges. Algorithm AX uses a group communication service (GCS) [95] with certain properties to provide membership and communication services to the groups of processors. We argue that these properties are basic and are provided by several group communication systems and specifications [23]. It also uses a coordinator-based approach for load-balancing the tasks within each group of processors. To analyze the algorithm we introduce *view-graphs* that are directed acyclic graphs used to represent the partially-ordered view evolution history witnessed by the processors (the group changes that processors undergo during the computation). We believe that view-graphs have the potential of serving as a general tool for studying cooperative computing with group communication services. We show that algorithm AX solves the *Omni-Do* problem for n tasks, p processors and any pattern of group fragmentations and merges with *task-oriented* work $W < \min\{nf_r + n, np\}$ and message complexity $M < 4(nf_r + n + pf_m)$, where f_r denotes the number of new groups created due to fragmentations and f_m the number of new groups created due to merges. This extends the work of Dolev, Segala and Shvartsman [32], mentioned in the previous section. In addition, algorithm AX has better message complexity (subquadratic in n) than the algorithm of Dolev *et al.* (at least quadratic in n) and the same asymptotic task-oriented work complexity, under group fragmentations.

An *Omni-Do* algorithm and its efficiency can only be partially understood through its worst case work analysis. This is because the resulting worst case bound might depend on unusual or extreme patterns of group reconfigurations where all algorithms perform poorly. In such cases, worst case work may not be the best way to compare the efficiency of algorithms. Hence, in order to understand better the practical implications of performing work in partitionable settings, we initiate the study of the *Omni-Do* problem as an on-line problem and we pursue *competitive analysis* [105]. Specifically, we study a simple randomized algorithm, called algorithm RS, where each processor (or group) determines the next task to complete by randomly selecting the task from the set of tasks this group does not know to be completed. We compare the expected task-oriented work of this algorithm to the task-oriented work of an "off-line" algorithm that has full knowledge of the future changes in the communication medium. We consider arbitrary patterns of network reconfigurations (including but not limited to fragmentations and merges). We describe a notion of *computation width*, which associates a natural number with a history of changes in the communication medium, and show both upper and lower bounds on competitiveness in terms of this quantity. Specifically, we show that algorithm RS obtains the competitive ratio $(1 + \mathbf{cw}/e)$, where $\mathbf{cw}$ is the computation width; we also show that this ratio is tight. We note that $\mathbf{cw}$ captures precisely the effect of network reconfigurations on the efficiency of the computation.

1.5 Document Structure

The rest of the thesis is organized as follows. In Chapter 2 we survey prior and related work. In Chapter 3 we formally define the models of computation, the *Do-All* problem and its variations, and the measures of efficiency we use to evaluate *Do-All* algorithms. In Chapter 4 we present matching failure-sensitive upper and lower bounds on work for *Do-All* and *iterative Do-All*. We consider the model with synchronous crash-prone processors that are assisted by an "oracle" for load-balancing and termination decisions (assumption of perfect knowledge). In Chapter 5 we present failure-sensitive bounds on work and messages for the

Do-All problem for synchronous message-passing processors prone to crashes. We first consider a message-passing model where reliable multicast is available (Section 5.1) and then we consider a message-passing model without reliable multicast (Section 5.2). In Chapter 6 we present failure-sensitive bounds on work for the *Write-All* problem for synchronous crash-prone processors, first in a shared-memory model where the memory access concurrency does not need to be controlled (Section 6.1), and then in a shared-memory model where the memory access concurrency must be controlled (Section 6.2). Chapter 7 considers the *Omni-Do* problem in partitionable networks. We first analyze algorithmic solutions to *Omni-Do* in terms of worst case work (Section 7.1), and then we analyze the work of *Omni-Do* algorithms in terms of competitive analysis (Section 7.2). We conclude in Chapter 8 with a discussion of future research directions.

Chapter 2

Related Work

In this chapter we overview results for *Do-All* in several models of computation. We also give an overview of *group communication services*, and two problems that are related to *Do-All*, the *cooperative collect* and *consensus* problems. We conclude this section with a discussion on *web-based computing*.

2.1 Do-All in Message-Passing Models

Dwork, Halpern and Waarts were the first to consider *Do-All* in message-passing systems [33]. They developed several deterministic algorithms that solved the problem for synchronous crash-prone processors. To evaluate the performance of their algorithms, they used the "total number of tasks performed" work complexity measure (task-oriented work), denoted by W and the "total number of messages sent" message complexity measure, denoted by M. They also used the *effort* complexity measure, defined as the sum of W and M. This measure of efficiency makes sense for algorithms for which the work and message complexities are similar, which was the case for the algorithms in [33]. One algorithm presented in [33], called protocol

$\mathcal{B}$ has effort $O(n+p\sqrt{p})$, with work contributing the cost $O(n+p)$ and the message complexity contributing the cost $O(p\sqrt{p})$ toward the effort. The running time of the algorithm is $O(n+p)$. The algorithm uses synchrony to detect processor crashes by means of timeouts. The algorithm operates as follows. The n tasks are divided into chunks and each chunk is divided into subchunks. Processors checkpoint their progress by multicasting the completion information to subsets of processors after performing a subchunk, and broadcasting to all processors after completing chunks of work. Another algorithm, called protocol $\mathcal{C}$ has effort $O(n + p \log p)$. It has optimal work $W = O(n + p)$, message complexity $M = O(p \log p)$ and time $O(p^2(n + p)2^{n+p})$. This shows that reducing the message complexity may cause a significant increase in time. The last algorithm presented in [33], called protocol $\mathcal{D}$, obtains work optimality and it is designed for maximum *speed-up* (the ratio between the parallel time over the sequential time), which is achieved with a more aggressive check-pointing strategy, thus trading-off time for messages. The message complexity is quadratic in p for the fault-free case, and in the presence of $f < p$ crashes the message complexity degrades to $\Theta(fp^2)$. Finally, the authors in [33] demonstrate how each of their algorithms can be used to construct efficient algorithms for the Byzantine agreement problem (see Section 2.7 for more details).

De Prisco, Mayer and Yung [28] provided an algorithmic solution for *Do-All* considering the same setting as Dwork *et al.*, (synchrony, processor crashes) but using the "available processor steps" work complexity measure, denoted by S. De Prisco *et al.* use a "lexicographic" criterion: first evaluate an algorithm according to its available processor steps and then according to its message complexity. This approach makes sense when optimization of work is more important than optimization of communication. They present a deterministic algorithm that has $S = O(n + (f + 1)p)$ and $M = O((f + 1)p)$. The algorithm operates as follows. At each step all the processors have a consistent (over)estimate of the set of all the available processors (using checkpoints). One processor is designated to be the coordinator. The coordinator allocates the undone tasks according to a certain load balancing rule and waits for notifications of the tasks which have been performed. The coordinator changes over time. To avoid a quadratic upper bound for S, substantial processor slackness is assumed ($p \ll n$). We note that f appears in the equations mainly because of the use of the coordinator approach. The authors also develop a lower bound of $S = \Omega(n + (f+1)p)$ for any algorithm using the stage-checkpoint strategy, this bound being quadratic in p for f comparable with p.

Galil, Mayer and Yung [44], while working in the context of Byzantine agreement (see Section 2.7) assuming synchronous crash-prone processors, developed an efficient algorithm that has the same work bound as De Prisco *et al.* [28] ($S = O(n + (f+1)p)$) but has better message complexity: $M = O(fp^{\varepsilon} + \min\{f+1, \log p\}p)$, for any $\varepsilon > 0$. The improvement on the message complexity is mainly due to the improvement of the checkpoint strategy used in [28] by replacing the "rotating coordinator" approach with what they called the "rotating tree" (*diffusion tree*) approach.

Chlebus, De Prisco and Shvartsman [17] developed the only known efficient deterministic algorithm, that solves *Do-All* in the synchronous model under processor crashes and restarts. Their algorithm, called AR, uses an algorithmic technique that is based on an aggressive coordination paradigm by which multiple coordinators may be active as the result of failures: when the failures of coordinators disrupt the progress of the computation, the number of coordinators is increased (doubled); when the failures recede, a single coordinator is chosen. Algorithm AR has work $S = O((n + p\log p + f) \cdot \min\{\log p, \log f\})$ and message complexity $M = O(n + p\log p + fp)$, where f is the number of processor crashes and restarts.

En route to the solution for restartable processors, the authors presented another algorithm, called AN, which is designed to solve *Do-All* for synchronous processors prone to crashes (no restarts). Algorithm AN has work $S = O((n + p \log p / \log\log p) \log f)$ and message complexity $M = O(n + p \log p / \log\log p + fp)$, where f is the number of processor crashes. Observe that algorithm AN has better work than the algorithms in [28] and [44] when n, p and f are comparable. However, algorithms AN and AR assume reliable multicast [60] (if a processor fails while multicasting a message, then either all non-faulty processors deliver the message or none do), whereas prior solutions use simple point-to-point messaging. In Section 5.1 we give a more detailed description of algorithm AN and we develop failure-sensitive bounds on the work and message complexities that demonstrate precisely how processor crashes affect the efficiency of the algorithm.

Chlebus and Kowalski [20] studied the *Do-All* problem for synchronous crash-prone processors with reliable multicast under a *weakly-adaptive linearly bounded* adversary: the adversary selects $f < c \cdot p$ $(0 < c < 1)$ crash-prone processors prior to the start of the computation, then any of these processors may crash at any time during the computation. They designed a randomized algorithm with expected combined work complexity and message complexity $S + M = O(n + p(1 + \log^* p - \log^*(p/n)))$. They also showed that the performance of their randomized algorithm is better than that of any deterministic algorithm in the same setting, where work $S = \Omega(p \log p / \log\log p)$ has to be performed.

Chlebus, Kowalski and Lingas [22] studied *Do-All* in the setting of broadcast networks where processors communicate over a multiple access channel [45], synchronized by a global clock. If exactly one processor broadcasts at a time, then the message is delivered to all processors. If more than one processor broadcasts then *collision* occurs and no message is delivered.

The authors provide randomized and deterministic solutions with and without collision detection, and for various size-bounded adversaries.

Chlebus, Gasieniec, Kowalski and Shvartsman [19] developed a deterministic algorithm that solves *Do-All* for synchronous crash-prone processors with combined work and message complexity $S + M = O(n + p^{1.77})$. This is the first algorithm that achieves subquadratic in p combined S and M for the *Do-All* problem for synchronous crash-prone processors. They present another deterministic algorithm that has work $S = O(n + p \log^2 p)$ against f-bounded adversaries such that $p - f = \Omega(p^\alpha)$ for a constant $0 < \alpha < 1$. They also show how to achieve $S + M = O(n + p \log^2 p)$ against a linearly-bounded adversary by carrying out communication on an underlying constant-degree network.

Recently, Kowalski and Shvartsman [75] considered the *Do-All* problem in asynchronous message-passing systems. Recall that the *Do-All* problem can be solved without any communication with work $S = \Theta(np)$ by an oblivious algorithm where each processor performs all tasks. The authors observe that it is not possible to obtain subquadratic (in n) work when the *message delay* d is substantial, e.g., $d = \Theta(n)$. Therefore, they pursue a *message-delay-sensitive* approach: The upper bounds on work and communication are given as functions of p, n, and d, the upper bound on message delays, however algorithms have no knowledge of d and they cannot rely on the existence of an upper bound on d. The authors present two families of asynchronous algorithms achieving, for the first time, subquadratic work as long as $d = o(n)$. The first, is a family of deterministic algorithms parameterized by a positive integer q and a list of q permutations on the set $[q] = \{1, ..., q\}$, where $2 < q < p < n$. It is shown that for any constant $\varepsilon > 0$ there is a constant q such that the corresponding algorithm has work $S = O(np^\varepsilon + pd[n/d]^\varepsilon)$ and message complexity $M = O(p \cdot S)$. The algorithms in this family are modeled after an algorithm of Anderson and Woll [7] (see next section), and use a list of q permutations is a similar way. The second family, is a family of deterministic and randomized algorithms, parametrized by a list of p permutations on the set $[p]$. The randomized algorithms have expected work $S = O(n \log p + pd \log(2 + n/d))$ and expected message complexity $M = O(np \log p + p^2 d \log(2 + n/d))$. It is shown that there exists a deterministic list of schedules such that the deterministic algorithm has work $S = O(n \log p + pd \log(2 + n/d))$ and message complexity $M = O(np \log p + p^2 d \log(2 + n/d))$. The authors also present the first delay-sensitive lower bound for *Do-All* in this setting, that helps explain the behavior of the their algorithms. Specifically, they show that any deterministic (resp. randomized) algorithm with p asynchronous processors and n tasks has work (resp. expected work) of $\Omega(n + pd \log_{d+1} n)$.

2.2 Write-All in Shared-Memory Models

Kanellakis and Shvartsman were the first to consider the *Write-All* problem [67]. They developed the best known deterministic synchronous algorithm, called W, that solves *Write-All* under processor crashes with work $S = O(n+p \log n \log p/ \log \log p)$ [67]. The algorithm uses binary trees of depth $O(\log n)$ for estimating the number of operational processors, the number of completed tasks (elements of the input array that have value 1) and for balancing the loads of the operational processors. In particular, the elements of the input array are associated with the leaves of a binary tree of depth $O(\log n)$, called the *progress tree*. The processors are initially distributed to the leaves of the progress tree where each of them performs a task and writes 1 to the corresponding tree location. Then the processors traverse the tree bottom-up recording the progress that it made. This gives an (under)estimate of the number of done tasks. The processors also traverse, bottom-up, a tree of depth $O(\log p)$, called the *processor enumeration tree* to estimate the number of operational processors. Using the two estimated values, the processors traverse the progress tree top-down until they reach to a leaf of the tree. This evenly distributes the operational processors onto undone tasks. The processors perform the task associated with the leaf they reached, and then traverse the progress tree up to the root to record the new progress. This is repeated until all tasks are performed. Observe that the bound on work for algorithm W does not include f, the number of processor crashes. In Section 6.1 we give a more detailed description of algorithm W and we present our failure-sensitive analysis of its work complexity.

Kedem, Palem, and Spirakis [72] performed an average case analysis of algorithm W [67] considering *random* processor crashes (each processor may crash with a fixed probability). They showed that algorithm W can solve the *Write-All* problem with expected time $O(\log p \log n)$ and expected work $O((p + n) \log n)$. This shows that algorithm W performs well under random failures. In the same paper, Kedem *et al.* developed a simple algorithm, called algorithm PS, which is a trivial modification of the straightforward pointer-doubling algorithm (PS is short for pointer shortcutting). The algorithm improves on the expected time of algorithm W while it obtains the same expected work complexity. Specifically, algorithm PS solves the *Write-All* problem under random failures with expected time $O(\log n)$ and expected work $O(n \log n)$.

Kanellakis, Michailidis and Shvartsman [66] developed a deterministic synchronous algorithm, algorithm KMS (called algorithm $\mathrm{W}^{\mathrm{opt}}_{\mathrm{CR/W}}$ in [66]) that solves *Write-All* under processor crashes while controlling the read and write memory access concurrency. The algorithm uses the same data structures as algorithm W to record the progress of the computation and to perform load balancing, and it uses two additional data structures to control the memory access concurrency: (a) *processor priority trees* are used to determine which processors are allowed to read or write each shared location that has to be accessed concurrently by more than one processor, and (b) *broadcast arrays* are used to disseminate values among readers and writers. The write concurrency, denoted ω, measures the redundant write memory accesses as follows: Consider a step of a synchronous parallel computation, where a particular location is written by $x \leq p$ processors. Then $x - 1$ of these writes are "redundant", because a single write should suffice. Hence, the write concurrency for this step is $x - 1$. The read concurrency, denoted ρ, is measured in a similar manner. Algorithm KMS has work $S = O(n + p \log^2 n \log^2 p / \log\log n)$, write concurrency $\omega \leq f$ and read concurrency $\rho \leq f \log n$, f being the number of crashes. Observe that although the bounds on the read and write concurrencies are given as a function of f, the bound on work is not given as a function of f. In Section 6.2 we give a more detailed description of algorithm KMS and we present a failure-sensitive analysis of its work complexity.

Algorithm V [15] is a variation of algorithm W that solves *Write-All* with synchronous restartable crash-prone processors. As in algorithm W, the processors use binary trees of depth $O(\log n)$ to perform load balancing. Restarted processors join the computation at a pre-defined phase. Algorithm V requires work $S = O(n + p \log^2 n + f \log n)$, where f is the number of processor crashes and restarts. Observe that since f can be arbitrarily large, the work of algorithm V might not be bounded by a function of n and p.

Anderson and Woll [7] developed the best deterministic asynchronous algorithm for *Write-All*. We call this algorithm AW^{T}. Algorithm AW^{T} has work $S = O(np^{\varepsilon})$, for arbitrary $0 < \varepsilon < 1$. The algorithm uses a q-ary tree, called *progress* tree to load balance processors to tasks (array elements) and a list of $q \leq p$ permutations of $[q]$, used in conjunction with processor identifiers to let the processors know in what order to traverse each of the q subtrees of each interior node in the progress tree. The work complexity does not account for the time required for these permutations to be computed; it is assumed that they are known before the execution of the algorithm. The authors of [7] provide a construction (exponential in q processing time) of permutations needed by their algorithm. Groote *et al.* [59] introduced a different approach that does not use permutation lists and hence no pre-processing is needed to construct such lists. They present an algorithm that has work $S = O(np^{\log(\frac{x+1}{x})})$ where $x = n^{\frac{1}{\log p}}$. The authors argue that their algorithm performs better than AW^{T} under practical circumstances where $p \ll n$, e.g., when $n = p^2$. Another practical algorithm, that does not require a precomputed set of permutations is algorithm X, developed by Buss, Kanellakis, Radge and Shvartsman [15]. Algorithm X is a special case of algorithm AW^{T}, where $q = 2$ and it has work $S = O(np^{0.59})$. Algorithm X can also be used to solve the *Write-All* problem for synchronous processors prone to crashes and restarts using work $S = O(np^{0.59})$.

Recently, Malewicz [82] developed a deterministic asynchronous algorithm for the *Write-All* problem that has work $S = O(n + p^4 \log n)$. This is the first asynchronous *Write-All* algorithm that has optimal work for a nontrivial number of processors ($p < (n/\log n)^{1/4}$), as opposed to all previously known deterministic algorithms that require as much as $\omega(n)$ work when $p = n^{1/c}$, for any fixed $c > 1$. The algorithm operates on *collision* detection: each processor has a collection of intervals of the input array and iteratively selects an interval to work on. The processor proceeds from one edge of the interval toward the other edge, executing the tasks associated with the cells in the interval. When processors "collide", meaning that they are allocated to the same input element, they exchange appropriate information and schedule their future work accordingly. The algorithm uses Test-And-Set instructions to detect collisions, as opposed to the previous algorithms that used only atomic Read/Write instructions.

Kedem, Palem, Raghunathan and Spirakis [71] showed that any crash-free execution of an algorithm designed to solve *Write-All* deterministically for $n = p$ with crash-prone processors requires time $\Omega(\log n)$ and work $\Omega(n \log n)$. Martel and Subramonian [86] extended these lower bounds for randomized algorithms. Specifically they showed that the lower bound on expected time and expected work on randomized algorithms for *Write-All* is $\Omega(\log n)$ and $\Omega(n \log n)$, for $n = p$, respectively (these lower bounds apply to both synchronous crash-prone and asynchronous processors). Martel, Park, and Subramonian [85] developed a randomized asynchronous algorithm for *Write-All* that matches the above lower bound on the expected work for randomized algorithms. Their algorithm proceeds as follows: the locations of the input array are viewed as n leaves of a binary tree that is $\Theta(\log n)$ deep (this is similar to the progress tree of algorithm X [15]). Initially all tree nodes are unmarked. Each processor selects a tree node at random. If the node v is a leaf node or if its children are marked, then node v is also marked. This is repeated until the root is marked.

Write-All algorithms can be used *iteratively* to simulate parallel algorithms formulated for synchronous failure-free processors (see the works of Kedem, Palem, and Spirakis [72], Kedem, Palem, Raghunathan, and Spirakis [71], Martel, Park, and Subramonian [85], Martel, Subramonian, and Park [87], and Shvartsman [104]). It was shown that the execution of a single n-processor step on p failure-prone processors does not exceed the complexity of solving a n-size instance of *Write-All* using p failure-prone processors. This commonly requires that (i) the individual processor steps are made idempotent (since they may have to be performed multiple times due to failures or asynchrony), and that (ii) a linear in the number of processors auxiliary memory is made available (to be used as a "scratchpad" and to store intermediate results). While the former can be solved with the help of an automated tool, e.g., a compiler, the latter requires sophisticated solutions because of the difficulty of (re)using the auxiliary memory due to "late writers" (i.e., processors that are slow and that unknowingly write stale values to memory). Examples of randomized solutions addressing these problems include the works of Aumann and Rabin [9] , and Kedem, Palem, Rabin, and Raghunathan [70]. Another important aspect of algorithm simulations is the use of an optimistic approach, where the computation may proceed for several steps assuming that all tasks assigned to active processors are successfully completed. Such approach was used by Kedem, Palem, Raghunathan and Spirakis in [71]. In some deterministic models optimal simulations are possible (as demonstrated by Shvartsman in [104]), however randomized solutions are able to achieve (expected) optimality for broader ranges of models and algorithms. An example of a practical implementation is discussed by Dasgupta, Kedem and Rabin in [25].

2.3 Do-All Under the Assumption of Perfect Knowledge

Kanellakis and Shvartsman [68] showed that *Do-All* can be solved using unit-time memory snapshots (equivalently assuming perfect knowledge – see below) for synchronous crash-prone processors with work $S = O(n + p \log p / \log \log p)$ for $f < p \leq n$ (f is the number of processor crashes). They showed that this bound is tight, by giving a matching lower bound. The authors also presented a matching lower and upper bound on work for *Do-All* assuming synchronous crash-prone and restartable processors. The bound is $S = \Theta(n + p \log p)$ for $p \leq n$ and any f, the number of processor crashes and restarts. This result also holds for the model of perfect knowledge with asynchronous processors, where a crash and restart event can be modeled as a delay.

The above bounds hold under the assumption that processors can read all memory in constant time (memory snapshots). However, it is not difficult to see that the memory snapshot assumption in shared-memory is equivalent to the assumption of perfect knowledge, where a deterministic omniscient oracle provides load-balancing and termination to the processors in constant time (information that can be obtained also in constant time in the memory snapshots model). Hence any result provided in the "memory snapshots model" holds trivially in the "perfect knowledge model".

2.4 Omni-Do in Partitionable Networks

Omni-Do was introduced and studied by Dolev, Segala and Shvartsman in [32]. They present the following results, under the assumption that $p = n$. (a) For the case of dynamic group changes, including fragmentations and merges, they show that the termination time of any on-line task assignment algorithm is greater than the termination time of an off-line task assignment algorithm (that has the knowledge of the dynamic group changes pattern) by a factor greater than $n/12$. (b) They present a load balancing algorithm, called AF that solves the *Omni-Do* problem with group fragmentations (no merges) and under the assumption that all processors belong initially to a single group, with work $S = O(n + f \cdot n)$, $f < n$ being the *fragmentation-number* of the computation. (The fragmentation-number of a fragmentation is the number of new groups created due to this fragmentation minus one. The fragmentation-number of the computation is the sum of all fragmentation-numbers of all the fragmentations occurred in the computation.) The basic idea of algorithm AF is the following: each processor performs undone tasks according to a certain load balancing rule until it learns the results of all tasks. The algorithm uses a group communication service to handle group memberships and communication within groups (see Section 2.5). The authors did not measure the message complexity of algorithm AF, however it is not difficult to see that M is at least quadratic. (c) They develop an effective scheduling strategy for minimizing the *task execution redundancy* (see below) between any two processors that merge during the computation. More specifically, they show that if initially all processors work in isolation, then the task redundancy incurred when the communication is first established between any two processors is bounded by 1 as long as no processor has executed more than $\Theta(n^{1/3})$ tasks. The task execution redundancy is defined as follows. Consider two processors, i and j, that at some point of the computation merge. Let T_i be the set of task identifiers of the tasks that processor i performed before the merge and let T_j be the set of task identifiers of the tasks that processor i performed before the merge. Let $R_i = T_i \cap T_j$. Then the task execution redundancy of this merge is $|R_i|$. (Hence, if processors i and j performed different tasks before they merge, then the task execution redundancy is zero.) We note that all the results in [32] were shown for the asynchronous timing model.

Malewicz, Russell, and Shvartsman in [83, 84] introduced the notion of k-*waste* that measures the redundant task-oriented work performed by k groups (or processors) when they start in isolation and then merge into a single group. However, they only adequately investigate the case of the *pairwise waste* (2-waste) until the *first merge*. This is the case when from a pool of p processors, any two processors merge into one group having performed a and b ($a, b \leq n$) tasks respectively. The authors first show a lower bound on pairwise waste of $\Omega(a^2/n)$ (when

$n \geq a \geq b$ and $n = p$). Then they present an asymptotically optimal randomized construction as well as a near-asymptotically-optimal deterministic construction using elements from design theory [63].

2.5 Group Communication Services

Group communication services (GCS) [95] provide membership and communication services to the group of processors. GCSs have been established as effective building blocks for constructing fault-tolerant distributed applications. These services enable the application components at different processors to operate collectively as a group, using the service to multicast messages. The basis of a group communication service is a *group membership service*. Each processor, at each time, has a unique *view* of the membership of the group. The view includes a list of the processors that are members of the group. Views can change and may become different at different processors. There is a substantial amount of research dealing with specification and implementation of GCSs. Some GCS implementations are Isis [14], Transis [30], Totem [91], Newtop [37], Relacs [10], Horus [110], Consul [88] and Ensemble [61]. Some GCS specifications are presented in [97, 11, 38, 31, 24, 62, 90]. An extended study on specifications of GCS can be found at [23]. Examples of recent work dealing with primary groups are [27, 77]. An example of an application using a GCS for load balancing is by Fekete, Khazan and Lynch [73]. Babaoglu *et al.* [12] study systematic support for partition awareness based on group communication services in a wide range of application areas, including applications that require load balancing. To evaluate the effectiveness of partitionable GCSs, Sussman and Marzulo [107] proposed a measure (*cushion*) precipitated by a simple partition-aware application.

2.6 Cooperative Collect

Omni-Do has an analogous counterpart in the shared-memory model of computation, called the *collect* problem, introduced by Shavit [103] and studied by Saks, Shavit and Woll in [100]. There are p processors each with a shared register. The goal is to have all the processors learn (collect) all the register values. Computation is asynchronous, with the adversary controlling timing of the processors. A trivial solution to this problem is to have all p processors reading all p registers. Saks, Shavit and Woll recognized the opportunity for improving the efficiency of shared-memory algorithms by finding a way for processors to cooperate during their collects [100]. They developed a randomized algorithm, which they analyzed in a model where a time unit is the minimal interval in the execution of the algorithm during which each processor executes at least one step (known as the *big-step* model). The goal is to minimize the number of big-steps.

Ajtai, Aspnes, Dwork and Waarts [3] showed that the problem can be solved deterministically with work $S = O(p^{3/2} \log p)$, by adapting the algorithm of Anderson and Woll (AW^{T}) [7]. The authors assume single-writer, multi-reader registers, each of size $O(p \log p)$ bits. The authors point out that for the asynchronous *Write-All* problem, usually some sort of multi-writer registers are assumed, each of size $O(\log p)$ bits. Then, the authors argue that for a model that provides multi-writer registers, the cooperative collect would be equivalent to the *Write-All* problem: given a *Write-All* algorithm, if each of the writes to the registers is replaced by a read, and the value read is propagated along with the certification that the particular register was accessed, then when each processor terminates, it knows that each register was accessed along with each register's value. The authors, using this observation, show that the algorithm of Anderson and Woll [7] can be modified to solve the collect problem using single-writer, multi-reader registers, by choosing "appropriate" set of permutations on $[p]$ that they show to exist.

Aspnes and Hurwood [8] developed a randomized algorithm for the cooperative collect problem that has work $S = O(n \log^3 n)$ with high probability. The idea of the algorithm is that each processor keeps reading randomly selected registers. However, before a processor attempts to read a register, it "leaves a note" saying where it is going. This is necessary to prevent situations where the adversary chooses a specific register and delay each processor that attempts to read that register (it is not difficult to see that this leads to quadratic work). The authors show that the processors, using the notes left by other processors, can detect such traps with high probability and hence avoid quadratic work with high probability. The work achieved by this algorithm is very close to the lower bound of $\Omega(n \log n)$, shown in [100].

Although the algorithmic techniques when dealing with the collect problem are different, the goal of having all processors to learn a set of values is similar to the goal of having all processor to learn the results of a set of tasks in *Omni-Do*.

2.7 Consensus

Consensus is the abstract problem of having p processors to agree on a common value. This problem is one of the fundamental problems of distributed computing, and solutions to this problems are used as building blocks in various distributed applications. Dwork, Halpern and Waarts [33] showed that algorithmic solutions to *Do-All* can be used to provide efficient solutions to consensus. Also, De Prisco, Mayer and Yung [28], and Galil, Mayer and Yung showed that algorithmic solutions to consensus can be used to solve *Do-All*. The consensus problem has been studied in various models of computation and we present an overview, focusing mainly on work that is related to our research.

The Coordinated Attack Problem

The *coordinated attack* problem is a fundamental problem of reaching consensus in message-passing systems, where messages may be lost. It was introduced by Gray [57] in the context of distributed databases. Abstractly, there are several generals that want to agree on an attack time, and that communicate using messengers who may be lost. Gray [57] proved that this problem is impossible to be solved deterministically in the absence of reliable communication, even if the system is synchronous. Due to this impossibility result, the randomized version of the coordinator attach problem has been considered: agreement is reached with high probability. Unlike the deterministic version, the randomized coordinated attack problem can be solved (in synchronous systems). See, for example, [111].

Byzantine Agreement and its Connection to Do-All

When processors are subject to processor failures (rather than communication failures), consensus is better known as *Byzantine agreement*. Byzantine agreement was introduced by Lamport, Shostak and Pease [76] in which consensus was formulated in terms of *Byzantine generals* prone to *Byzantine* failures (faulty processors may exhibit totally unconstrained behavior [76]): as in the coordinated attack problem, the generals want to agree on a time to carry out an attack, but in this case, they do not worry about lost messengers, but about the traitorous behavior by some general. Alternatively, the problem is formulated, for both crash and Byzantine failures, as follows: p processors, a subset of which may be faulty, must agree on a value broadcast by a distinguished processor called the *sender* or the *general* in such a way that all non-faulty processors decide the same value, and when the general is non-faulty, they decide on the value the general sent. The number of faulty processors is bounded in advance, by a fixed number f.

For the synchronous message-passing model with Byzantine processor failures, Pease, Shostak and Lamport [94, 76] presented upper and lower bounds of $3f + 1$ for the number of processors required for Byzantine agreement. Moses and Waarts [92], Berman and Garay [13] and Garay and Moses [46] have produced $f + 1$ *round* Byzantine agreement algorithms (in each round, each processor can send messages to other processors and receive the messages sent by other processors in the same round) with polynomial communication (number of point-to-point messages sent). Fischer and Lynch [40] showed that Byzantine agreement cannot be solved in fewer than $f + 1$ rounds.

Byzantine agreement was also studied in the synchronous message-passing model under processor crashes. In [80] two deterministic algorithms are presented that solve Byzantine agreement (each using a different technique) in $f + 1$ rounds and with $O((f + 1)p^2)$ message complexity, where p is the number of processors and f the number of processor crashes. For the same model, Dwork and Moses [35] showed that Byzantine agreement cannot be solved in fewer than $f + 1$ rounds (like in the case of Byzantine failures).

Dwork, Halpren and Waarts [33], while working in the context of the *Do-All* problem assuming synchronous crash-prone processors (see Section 2.1), developed an algorithm that can use a *Do-All* algorithm as a building block to solve the Byzantine agreement problem for synchronous crash-prone processors. Their algorithm proceeds in two stages: first the general broadcasts its value to processors with $\text{PID} = 1, \ldots, f + 1$. Then these $f + 1$ processors use one of the *Do-All* algorithms (Protocols $\mathcal{B}$, $\mathcal{C}$ or $\mathcal{D}$ [33]) to perform the "work" of informing processors $1, \ldots p$ about the general's value. Hence, performing a *Do-All* task here means sending a message containing the general's value. Initially all processors have the initial value 0 as the general's value (the general of course has it own value as initial value). When a processor receives a message about a value for the general different from its current value, it adopts the new value. Finally, at a predetermined time by which the underlying *Do-All* algorithm is guaranteed to have terminated, each processor decides on its current value for the general. Using protocol $\mathcal{C}$ as the *Do-All* algorithm the authors solve the Byzantine agreement problem for synchronous crash-prone processors in $O(2^p)$ time and with $O(p + f \log f)$ message complexity. When they use protocol $\mathcal{B}$ they obtain a Byzantine agreement solution of $O(p)$ time and $O(p + f\sqrt{f})$ message complexity. Observe that when p and f are comparable, the second solution has the same asymptotic time complexity as the algorithms presented in [80] and substantially better message complexity. This shows that *Do-All* solutions can yield efficient solutions to the Byzantine agreement problem (and to the consensus problem in general).

Galil, Mayer and Yung [44] developed an algorithm that solves Byzantine agreement for synchronous crash-prone processors that uses a linear number of messages ($O(p)$) and super-linear time ($O(p^{1+\varepsilon})$). They also improved the message complexity of the *Do-All* algorithm of De Prisco *et al.* [28] (see Section 2.1). This algorithm relies on two agreement-like protocols: (a) the check-point protocol that processors use to agree on the set of operational processors, and (b) the synchronization protocol that processors use to agree on the time that the next check-point protocol will begin. Given the full details of the protocols, it is not difficult to observe that these protocols solve multiple instances of the Byzantine agreement problem. This shows that efficient solutions to consensus can lead to efficient solutions to *Do-All*.

FLP Impossibility Result

One of the fundamental impossibility results in the theory of distributed computing is the FLP result which states that consensus cannot be solved in asynchronous models, even if there is guaranteed to be no more than one processor failure. More precisely, every asynchronous consensus algorithm has the possibility of nontermination (that is, a non-faulty processor might never decide on a value and the algorithm runs indefinitely), even with only one faulty processor. This result was shown by Fischer, Lynch and Paterson (thus the name FLP) in [41] for the asynchronous message-passing model and it was later extended to the read/write asynchronous shared-memory model by Loui and Abu-Amara [78]. Since this impossibility result has practical implications for distributed applications in which agreement is required, a lot of research has been done in solving consensus in asynchrony either by relying on randomized correctness or by weakening the problem (e.g. k-Agreement, approximate agreement) or strengthening the model (e.g. assuming read-modify-write or compare-and-swap shared memory, using failure detectors, introducing some timing conditions — partial synchrony). For such solutions we refer the reader to [34, 16, 80].

The research performed on consensus in the models that allow fault-tolerant solutions teach that the maximum number of processor failures needs to be included in upper/lower bounds and impossibility results. This contributes to the understanding of the impact that failures have on the efficiency and dependability of algorithms and in identifying the trade-offs between fault-tolerance and efficiency for solving distributed problems (such as *Do-All*). This research motivated in part the research in this thesis: show failure-sensitive upper/lower bounds for the *Do-All* problem.

2.8 Web-Based Computing

In recent years, the web has become the computing platform of choice for a variety of computational problems that cannot be handled efficiently by the traditional fixed-size collection of machines (such as clusters of workstations or multiprocessor machines). This has given rise to the study of *web-based computing* (WBC) [99]: A large number of processing elements cooperate in computing a large number of independent tasks. A usual WBC computation proceeds as follows: Interested "volunteers" register with a specific web-site. Then, each registered volunteer visits the web-site occasionally to receive a task to compute. Once the volunteer performs the task, it returns the results from that task. The computation continues in this manner.

Possibly the most popular web-based project is SETI@home [74]. SETI stands for "Search of Extra-Terrestrial Intelligence". The project, initiated at University of California at Berkeley, was the first attempt to use large-scale distributed computing to perform a search for radio signals possibly coming from extraterrestrial civilizations. It soon became obvious that great amount of computer power would be necessary to get the job done: the universe is potentially infinite, and the parameters of a possible alien signal are unknown. The SETI team counts on using thousands of home personal computers that are idle most of the time, especially when their owners are at work or are asleep. People, can register at the project's web-site (http://setiathome.ssl.berkeley.edu), and make their computer available to the project, when they are not using it.

Several web-based projects similar to SETI@home are in existence. For example, the RSA@home project [96]. The project is involved in finding the prime factors of large integers. The problem of factoring integers has drawn considerable attention due to the RSA cryptographic scheme [98], since the security of RSA depends upon the difficulty of factoring large numbers. Another example is the AIDS@home project [93]. Through the "donation" of large computing power, scientists and researchers have an ideal system to model the evolution of drug resistance and design anti-HIV drugs necessary to fight AIDS. More examples of web-based projects can be found at http://www.intel.com/cure.

As we demonstrate later on (see Section 4.3), complexity results obtained for *Do-All* in the model of perfect knowledge can yield insight about the bounds on task execution redundancy in settings where a server repeatedly allocates tasks to failure-prone processors (as in web-based computing). This follows from the observation that the oracle assumed in the model of perfect knowledge can be used to abstract the server that makes the load-balancing decisions in web-based computing.

Chapter 3

Models of Computation and the Do-All Problem

In this chapter we define the models of computation, the *Do-All* problem and the efficiency measures we use to evaluate *Do-All* algorithms.

3.1 General Setting and Definitions

Distributed setting: We consider a distributed system consisting of p processors; each processor has a unique identifier (PID) from the set $\mathcal{P} = [p] = \{1, 2, \ldots, p\}$. We assume that p is fixed and is known to all processors.

Each processor's activity is governed by a local clock. When the processor clocks are assumed to be globally synchronized, our distributed setting is *synchronous* and we say that the processors are synchronous. In this case, processor activities are structured in terms of synchronous *steps* (constant units of time). When the processors take steps at arbitrary relative speeds, our distributed setting is *asynchronous* and we say that the processors are asynchronous.

Tasks: We define a *task* to be any computation that can be performed by a single processor in constant time. The tasks are assumed to be *similar*, *independent*, and *idempotent*. By the similarity of the tasks we mean that the task executions consume equal or comparable resources. By the independence of the tasks we mean that the tasks can be executed in any order, that is, the execution of a task is independent of the execution of any of the other tasks. By the idempotence of the tasks we mean that executing a task many times and/or concurrently has the same effect as executing the task once. We define the *result* of a task to be the outcome of the task execution. Each task has a unique identifier (TID) from the set $\mathcal{T} = [n] = \{1, 2, \ldots, n\}$. We assume that n is fixed and known to all processors.

We also consider sequences of task-sets $\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_r$, where each $\mathcal{T}_i$, for $1 \leq i \leq r$, is a set of n tasks and the execution of any task in $\mathcal{T}_i$ must be delayed until all tasks in $\mathcal{T}_{i-1}$ are performed. This models the situation where the execution of the tasks in $\mathcal{T}_i$ depends on the execution of the tasks in $\mathcal{T}_{i-1}$, for $2 \leq i \leq r$. However we assume that the tasks within each $\mathcal{T}_i$ are independent, similar and idempotent and that they are known to all processors. We also assume that each task in $\mathcal{T}_i$, $1 \leq i \leq r$, has a unique TID. For example, each task in $\mathcal{T}_i$ may have a TID from the set $\{(i-1)n + 1, (i-1)n + 2, \ldots, in\}$.

Communication: In message-passing models, processors communicate by sending messages. Unless otherwise stated (see partitionable networks below), the underlying communication network is assumed to be fully connected, that is, any processor in $\mathcal{P}$ can send messages to any other processor in $\mathcal{P}$. We also assume that messages are neither lost nor corrupted in transit. In partitionable networks, the processors may be partitioned into groups of communicating processors. We assume that communication within groups is reliable but communication across groups is not possible. Partitions may change over time.

In synchronous message-passing systems we assume that message delivery has fixed known latency. Specifically, within a step, a processor can send messages to other processors and receive messages from other processors sent to it in the previous step (if any). In asynchronous systems, we assume no bounds on the message delivery latency.

In shared-memory models, processors communicate by reading from and writing to shared-memory locations. We assume that it takes a unit of time for a processor to read or write to a memory cell, according to its local clock. We consider synchronous shared-memory systems where the reads and writes can be concurrent. When two or more processors simultaneously write to the same memory cell, either *common* or *arbitrary* concurrent write discipline is observed. This follows the conventions established for the Parallel Random Access Machine (PRAM) [43]: for the common writes it is assumed that all values concurrently written to a memory location are the same, and for the arbitrary writes it is assumed that the concurrent writes to the memory location are arbitrarily ordered.

The assumption of perfect knowledge: In Chapter 4 we consider computations where the processors, instead of communicating with each other, communicate with some deterministic omniscient *oracle*, call it oracle $\mathcal{O}$, to obtain information regarding the status of the computation. In particular, the oracle informs the processors whether the computation is completed and if not, what task to perform next. We assume that the oracle performs perfect load-balancing, that is, the live processors are only allocated to unperformed tasks, and all such tasks are allocated a balanced number of live processors. We also assume that a processor can obtain load-balancing and termination information from the oracle in $O(1)$ time and that it can consult the oracle only once per local clock-tick.

The assumption of perfect knowledge (or the oracle assumption) abstracts away any concerns about communication that normally dominate specific message-passing and shared-memory models. This allows for the most general results to be established and it enables us to use these results in the context of specific models by understanding how the information provided by an oracle is simulated in specific algorithms. Also, any lower bound developed under the assumption of perfect knowledge, applies equally well to message-passing or shared-memory models.

3.2 Models of Adversity

In this section we present the models of adversity. We first present the failure types and then we introduce the notion of an adversary and present specific adversarial models.

3.2.1 Failure Types

We consider the following failure types.

Processor stop-failures/crashes ([102])**:** We consider *crash* failures, where a processor may crash at any moment during the computation and once crashed it does not restart. For message-passing models we assume that messages sent to crashed processors are lost and no messages are sent by crashed processors. For shared-memory models we assume that no reads and writes are performed by crashed processors. We also assume that processor crashes do not corrupt the contents of the shared-memory or make the shared-memory inaccessible. Following [102], we define a *fail-stop* failure to be a crash failure that can be detected. In synchronous settings, crash failures can be detected (by timeouts) and hence in such settings the two terms have the same meaning.

Regroupings/partitionable networks (applicable only to message-passing systems): We consider *partitionable networks* where dynamic changes to the network topology partition the processors into non-overlapping *groups* of communicating processors (processors do not crash). We represent each processor group g as a pair $\langle g.id, g.set \rangle$, where $g.id$ is the unique identifier of g and $g.set$ is the set of processor identifiers that constitute the membership of the group. To reduce notation clutter, for this point on, given a group named g we use g to stand for $g.set$ (e.g., if two, possibly distinct, groups g and g' have identical membership, we express this by $g = g'$). We refer to a transition from one partition to another as a *regrouping*. We also consider special types of regroupings: when a single group partitions into a collection of new groups, we call this a *fragmentation*. When a collection of groups merge and form a new group that contains all the processors of the merging groups, we call this a *merge*.

3.2.2 Adversarial Models

The concept of the *adversary* is useful for obtaining lower bound results for specific problems. An event caused by the adversary, e.g., a processor crash, in a computation may negatively affect the efficiency of the computation. We consider two *adversary types*:

(a) *omniscient and on-line*: the adversary has complete knowledge of the computation that it is affecting, and it makes instant dynamic decisions on how to affect the computation.

(b) *oblivious and off-line*: the adversary determines the sequence of events it will cause before the start of the computation and without having any *a priori* knowledge on how the computation will be affected under this sequence.

Note that the distinction between the two adversary types is only useful when considering randomized algorithms, where the knowledge or not of the random "coin tosses" may be significant. For deterministic algorithms the two adversary types are essentially the same, since the adversary knows exactly, before the beginning of the computation, how a specific deterministic algorithm would be affected by a specific event caused by the adversary.

Consider an adversary $\mathcal{A}$ and an algorithm Λ that solves a specific problem under adversary $\mathcal{A}$. We denote by $\mathcal{E}(\Lambda, \mathcal{A})$ the set of all executions of algorithm Λ for adversary $\mathcal{A}$. Let ξ be an execution in $\mathcal{E}(\Lambda, \mathcal{A})$. We denote by $\xi|_{\mathcal{A}}$ the set of events caused by $\mathcal{A}$ in ξ and we refer to it as the *adversarial pattern* of ξ. For an adversarial pattern $\xi|_{\mathcal{A}}$ of an execution ξ, we denote by $||\xi|_{\mathcal{A}}||$ the *weight* of $\xi|_{\mathcal{A}}$. The value of $||\xi|_{\mathcal{A}}||$ depends on the specific adversary $\mathcal{A}$ considered (e.g., if adversary $\mathcal{A}$ causes processor crashes, then $||\xi|_{\mathcal{A}}||$ is the number of crashes caused by the adversary; if the adversary causes fragmentations, then $||\xi|_{\mathcal{A}}||$ is the number of new groups created due to the fragmentations). Unless otherwise stated, we assume that the processors have knowledge neither of $\xi|_{\mathcal{A}}$ nor of any bounds on $||\xi|_{\mathcal{A}}||$.

We now present the adversaries we consider in the thesis. We first present the adversaries that cause processor failures and then we present the adversaries that cause regroupings.

Adversaries Causing Processor Failures

We consider only one adversary that causes processor failures. In particular, we consider an adversary that causes processor crashes.

Adversary $\mathcal{A}_S$: We denote by $\mathcal{A}_S$ an omniscient and on-line adversary that can cause processor crashes (but not restarts). Consider an algorithm Λ that solves a problem under adversary $\mathcal{A}_S$. Let ξ be an execution in $\mathcal{E}(\Lambda, \mathcal{A}_S)$. Then, the adversarial pattern $\xi|_{\mathcal{A}_S}$ is a set of triples (*crash*, PID, t), where crash is the event caused by the adversary, PID is the identifier of the processor that crashes, and t is the time of the execution (according to some external clock not available to the processors) in which the adversary forced processor PID to crash. Note that any adversarial pattern contains at most one triple (*crash*, PID, t) for any PID, i.e., if processor PID crashes, time t during which it crashes is uniquely defined.

For an adversarial pattern $\xi|_{\mathcal{A}_S}$ we define $\|\xi|_{\mathcal{A}_S}\|$ to be the number of processors that crash. For the purpose of the thesis we consider only executions ξ where $\|\xi|_{\mathcal{A}_S}\| < p$, that is we require that the adversary leaves at least one processor operational in the entire course of the computation to ensure computational progress.

Adversaries Causing Regroupings

We consider three adversaries that cause regroupings. The first one is an omniscient and on-line adversary that can cause only fragmentations and the second one is an omniscient and on-line adversary that can cause fragmentations and merges. The third one is an oblivious and off-line adversary that can cause arbitrary regroupings. This adversary is assumed to be oblivious and off-line because later in the thesis we consider randomized algorithms under this adversary, as opposed to the first two adversaries where we consider deterministic algorithms (this is also the case for adversary $\mathcal{A}_S$).

Adversary $\mathcal{A}_F$: We denote by $\mathcal{A}_F$ an omniscient and on-line adversary that can cause only group fragmentations (no merges). Consider an execution ξ of an algorithm Λ that solves a specific problem under $\mathcal{A}_F$, i.e., $\xi \in \mathcal{E}(\Lambda, \mathcal{A}_F)$. For the purpose of this thesis we consider only executions where initially all processors belong in a *single* group.

When adversary $\mathcal{A}_F$ forces group g to fragment into groups $g_1, g_2, \ldots, g_k$ we require that (a) $\bigcup_{i\in[k]} g_i = g$, and (b) $\forall i, j$ s.t. $1 \leq i, j \leq k$ and $i \neq j$, $g_i \cap g_j = \emptyset$. We say that the *fragmentation-number* of this fragmentation is k. Note that k new groups are created due to this fragmentation. Syntactically, we present such fragmentations in the adversarial pattern $\xi|_{\mathcal{A}_F}$ as the triple $(\textit{fragmentation}, g, \{g_1, g_2, \ldots, g_k\})$. Consequently, we represent an adversarial pattern $\xi|_{\mathcal{A}_F}$ of an execution ξ as a set of such triples and we define the fragmentation-number $f_r(\xi|_{\mathcal{A}_F}) = \|\xi|_{\mathcal{A}_F}\|$ to be the sum of the fragmentation-numbers of all the fragmentations in $\xi|_{\mathcal{A}_F}$. In other words, $f_r(\xi|_{\mathcal{A}_F})$ is the total number of new groups created due to the fragmentations in $\xi|_{\mathcal{A}_F}$. By convention, when a group is regrouped in such a way that it forms a new group with the same participants, we view this as a fragmentation.

Adversary $\mathcal{A}_{FM}$: We denote by $\mathcal{A}_{FM}$ an omniscient and on-line adversary that can can cause fragmentations *and* merges. Consider an execution ξ of an algorithm Λ that solves a specific problem under $\mathcal{A}_{FM}$, i.e., $\xi \in \mathcal{E}(\Lambda, \mathcal{A}_{FM})$. As for adversary $\mathcal{A}_F$, we consider only executions where initially all processors belong in a single group.

When adversary $\mathcal{A}_{FM}$ forces groups $g_1, g_2, \ldots, g_\ell$ to merge and form a group g, we require that $g = \bigcup_{i\in[\ell]} g_i$, and we say that the *merge-number* of this merge is 1 (note that a merge results to the creation of only one new group). Syntactically, we present such a merge in the adversarial pattern $\xi|_{\mathcal{A}_{FM}}$ as the triple $(\textit{merge}, \{g_1, g_2, \ldots, g_\ell\}, g)$. Fragmentations are presented as for adversary $\mathcal{A}_F$. Therefore, we represent an adversarial pattern $\xi|_{\mathcal{A}_{FM}}$ of an execution ξ as a set of "fragmentation" and "merge" triples, and we define the merge-number $f_m(\xi|_{\mathcal{A}_{FM}})$ to be the sum of all merge-numbers of all merges in $\xi|_{\mathcal{A}_{FM}}$. Then, $\|\xi|_{\mathcal{A}_{FM}}\| = f_r(\xi|_{\mathcal{A}_{FM}}) + f_m(\xi|_{\mathcal{A}_{FM}})$. In other words, $\|\xi|_{\mathcal{A}_{FM}}\|$ is the total number of new groups created, due to the fragmentations and merges in $\xi|_{\mathcal{A}_{FM}}$.

Observe that adversary $\mathcal{A}_{FM}$ is more powerful than $\mathcal{A}_F$, and that $\mathcal{E}(\Lambda, \mathcal{A}_F) \subseteq \mathcal{E}(\Lambda, \mathcal{A}_{FM})$ for an algorithm Λ that solves a specific problem. Also note that since we consider only executions ξ where all processors initially belong in a single group (and from the convention mentioned in the description of adversary $\mathcal{A}_F$ regarding a group being formed by a group with the same members), we have that $f_r(\xi|_{\mathcal{A}_{FM}}) > f_m(\xi|_{\mathcal{A}_{FM}})$.

Adversary $\mathcal{A}_{GR}$: We denote by $\mathcal{A}_{GR}$ an oblivious and off-line adversary that can cause arbitrary regroupings. Consider an algorithm Λ that solves a specific problem under adversary $\mathcal{A}_{GR}$. The adversary determines a sequence of regroupings prior to the start of an execution and it can not change this sequence once the execution has begun. We refer to such a predetermined sequence of regroupings as a *computation template*.

Adversary $\mathcal{A}_{GR}$ is restricted in determining only computations templates that can be expressed as the following labeled directed acyclic graph (DAG) $C = (V, E)$, which we call (p)-DAG (p is the number of participating processors): each vertex corresponds to a group of processors and a directed edge is placed from group g_1 to group g_2 if g_2 is created by a regrouping involving g_1. Each vertex of the DAG is labeled with the group of processors associated with that vertex. To this respect, the DAG is augmented with a labeling function $\gamma : V \rightarrow 2^{[p]} \setminus \{\emptyset\}$ (i.e., $\gamma(v)$ is the set of PIDs of the processors that belong in the group corresponding to vertex v). The function γ satisfies the following two conditions: (a) $[p] = \dot{\bigcup}_{v:\, indegree(v)=0}\gamma(v)$, and (b) there is a function $\phi : E \rightarrow 2^{[p]} \setminus \{\emptyset\}$ so that for each $v \in V$ with $indegree(v) > 0$, $\gamma(v) = \dot{\bigcup}_{(u,v)\in E}\phi((u, v))$, and for each $v \in V$ with $outdegree(v) > 0$, $\gamma(v) = \dot{\bigcup}_{(v,u)\in E}\phi((v, u))$. Here $\dot{\bigcup}$ denotes disjoint union. Note that the above definition allows for *several* initial groups (no more than p).

Given a (p)-DAG, we say that two vertices (groups) are *independent* if there is no direct path connecting one to the other. Then, for a computation template C, we define the *computation width of* C, $\mathbf{cw}(C)$ to be the maximum number of independent groups reachable (along directed paths) in the (p)-DAG, that represents C, from any vertex. This discussion is revisited in Section 7.2.1 where we give a formal definition of $\mathbf{cw}(C)$ using elements from set-theory and graph-theory.

Consider a problem of a specific size and all algorithms that solve this problem using the same number of processors, under adversary $\mathcal{A}_{GR}$. The same computation template can be applied to all these algorithms, however, the resulting execution might be different, depending on the steps that each algorithm takes in the presence of this computation template. Let C be a computation template determined by the adversary and let ξ be the resulting execution of an algorithm under this computation template. Note that the execution might terminate (meaning that the specific problem is solved) before all regroupings specified by the computation template take place (since the adversary does not know *a priori* how the algorithm would behave under this sequence of regroupings). Therefore, if (p)-DAG represents the computation template C, then the adversarial pattern $\xi|_{\mathcal{A}_{GR}}$ is represented by a subgraph of (p)-DAG. Furthermore, the weight of $\xi|_{\mathcal{A}_{GR}}$ is the computation width of this subgraph. Hence, $\|\xi|_{\mathcal{A}_{GR}}\| \leq \mathbf{cw}(C)$. For the purpose of this thesis, when considering algorithms under adversary $\mathcal{A}_{GR}$, failure-sensitivity is measured in terms of the properties of the computation templates. However, the efficiency of algorithms is measured based on the resulting executions.

3.3 The *Do-All* Problem

We now define the abstract problem of having p processors cooperatively perform n tasks in the presence of adversity.

Definition 3.1 *Do-All*: Given a set $\mathcal{T}$ of n tasks, perform all tasks using p processors, under adversary $\mathcal{A}$.

We let *Do-All*$_{\mathcal{A}}(n, p, f)$ stand for the *Do-All* problem for n tasks, p processors and adversary $\mathcal{A}$ constrained to adversarial patterns of weight less or equal to f. We consider *Do-All*$_{\mathcal{A}}(n, p, f)$ to be solved when all n tasks are completed and at least one operational processor knows about it. We let *Do-All*$_{\mathcal{A}}^{\mathcal{O}}(n, p, f)$ stand for the *Do-All*$_{\mathcal{A}}(n, p, f)$ problem when the processors are assisted by oracle $\mathcal{O}$ (as discussed in paragraph *Assumption of perfect knowledge* in Section 3.1).

In the shared-memory model the *Do-All* problem is known as the *Write-All* problem. The main difference is that in *Do-All* the tasks may be supplied to the processors from some external sources, while in *Write-All* the tasks are stored in shared-memory accessible to all processors. In the context of this thesis we abstract away from the sources and the nature of the tasks and we treat *Do-All* and *Write-All* as the same problem in that regard. However, when we study *Do-All* in shared-memory models, we will be referring to the *Write-All* problem, defined formally as follows:

Definition 3.2 *Write-All*: Given a zero-valued shared array of n elements, write the value 1 into each array location using p processors, under adversary $\mathcal{A}$.

We note that each "*Do-All* task" is associated with each location of the input array. When a processor sets the value of a certain location of the input array to 1, this implies that the processor has performed the associated task.

We let *Write-All*$_{\mathcal{A}}(n,p,f)$ stand for the *Write-All* problem for a shared array of n elements (or of n tasks), p processors and adversary $\mathcal{A}$ constrained to adversarial patterns of weight less or equal to f. We consider *Write-All*$_{\mathcal{A}}(n,p,f)$ to be solved, when the value of each of the n array elements is set to 1 (meaning that all tasks are performed) and at least one operational processor knows about it.

Do-All algorithms have been used in developing *simulations* of failure-free algorithms on failure prone processors [72, 104, 68]. This is done by iteratively using a *Do-All* algorithm to simulate the steps of the n failure-free "virtual" processors on p failure-prone "physical" processors (here the usual case is that the number of physical processors does not exceed the number of virtual processors, i.e., $p \leq n$). We abstract this idea as the *iterative Do-All* problem:

Definition 3.3 *r-Iterative Do-All*: Given any sequence $\mathcal{T}_1, \ldots, \mathcal{T}_r$ of r sets of n tasks, perform all $r \cdot n$ tasks using p processors by doing one set at a time, under adversary $\mathcal{A}$.

We let r-*Do-All*$_{\mathcal{A}}(n,p,f)$ stand for the *iterative Do-All* problem for r sets of n tasks, p processors and adversary $\mathcal{A}$ constrained to adversarial patterns of weight less or equal to f. We consider r-*Do-All*$_{\mathcal{A}}(n,p,f)$ to be solved, when all $r \cdot n$ tasks are completed and at least one operational processor knows about it. We let r-*Do-All*$_{\mathcal{A}}^{\mathcal{O}}(n,p,f)$ stand for the r-*Do-All*$_{\mathcal{A}}(n,p,f)$ problem when processors are assisted by oracle $\mathcal{O}$. The *r-Iterative Write-All* problem is defined similarly and it is denoted as r-*Write-All*$_{\mathcal{A}}(n,p,f)$.

When solving *Do-All* in partitionable networks, our goal is to utilize the resources of every group of the system during the entire computation. This is so for two reasons: (a) a client, at any point of the computation, may request for a result of a task from a certain group. This might be the only group that the client can communicate with. Hence, we would like all groups to be able to provide the results of all tasks, and (b) if different groups happen to perform different tasks and a regrouping merges these two groups, then more computational progress can be achieved with less computation waste. Hence, we would like all components to be computing in anticipation of regroupings.

Therefore, in partitionable networks, each processor must be computing until it learns the results of all tasks. We call this variation of *Do-All*, *Omni-Do*.

Definition 3.4 *Omni-Do*: Given a set $\mathcal{T}$ of n tasks and p message-passing processors, each processor must learn the result of all tasks, under adversary $\mathcal{A}$.

We let *Omni-Do*$_{\mathcal{A}}(n, p, f)$ stand for the *Omni-Do* problem for n tasks, p processors and adversary $\mathcal{A}$ constrained to adversarial patterns of weight less or equal to f. (For adversary $\mathcal{A}_{GR}$ we consider computation templates with computation width less or equal to f.) We consider *Omni-Do*$_{\mathcal{A}}(n, p, f)$ to be solved when all operational processors know the results of all n tasks.

Finally, we assume that the number of processors p is no more than the number of tasks n ($p \leq n$). Studying *Do-All* in the case of $p > n$ is not as interesting. This is so for two reasons: (1) the most interesting challenge is to consider the settings where maximum parallelism can be extracted for the case when each processor can initially have at least one distinct task to work on, (2) additionally, for the simulation results the most interesting case is when the number of simulating processors does not exceed the number of simulated processors.

3.4 Measures of Efficiency

We now define the complexity measures that will determine the efficiency of algorithms.

Work complexity. We first define the notion of *work*. We are considering two versions of the definition of work. The first definition of work, denoted by S, is based on the *available processor steps* measure, introduced by Kanellakis and Shvartsman in [67]. The second definition of work, denoted by W, is based on the *number of tasks performed* measure, introduced by Dwork, Halpern and Waarts in [33]. We note that the second definition is meaningful only for task-performing algorithms, while the first one is more general.

We assume that it takes a unit of time for a processor to perform a unit of work, according to its local clock. Let Λ be an algorithm that solves a problem of size n with p processors under adversary $\mathcal{A}$. For an execution $\xi \in \mathcal{E}(\Lambda, \mathcal{A})$ denote by $S_i(\xi)$ the number of processors completing a unit of work at time i of the execution, according to some external clock not available to the processors (for synchronous computations, the external clock is assumed to run in synchrony with the processors' local clocks).

Definition 3.5 (available processor steps or *work*) Let Λ be an algorithm that solves a problem of size n with p processors under adversary $\mathcal{A}$. If execution $\xi \in \mathcal{E}(\Lambda, \mathcal{A})$, where $\|\xi|_{\mathcal{A}}\| \leq f$, solves the problem by time $\tau(\xi)$ (according to the external clock), then the *work complexity* S of algorithm Λ is:

$$S = S_{\mathcal{A}}(n, p, f) = \max_{\xi \in \mathcal{E}(\Lambda, \mathcal{A}),\ \|\xi|_{\mathcal{A}}\| \leq f} \left\{ \sum_{i=1}^{\tau(\xi)} S_i(\xi) \right\}.$$

Note that in Definition 3.5 the idling processors consume a unit of work per idling step even though they do not contribute to the computation.

Let Λ be a task-performing algorithm that solves a problem with n tasks and p processors under adversary $\mathcal{A}$. For an execution $\xi \in \mathcal{E}(\Lambda, \mathcal{A})$ denote by $W_i(\xi)$ the number of processors completing a task-oriented unit of work (a task-oriented unit of work is a unit of work that is spent in performing a task) at time i of the execution, according to some external clock not available to the processors (for synchronous computations, the external clock is assumed to run in synchrony with the processors' local clocks).

Definition 3.6 (number of tasks performed or *task-oriented work*) Let Λ be a task-performing algorithm that solves a problem with n tasks and p processors under adversary $\mathcal{A}$. If execution $\xi \in \mathcal{E}(\Lambda, \mathcal{A})$, where $\|\xi|_{\mathcal{A}}\| \leq f$, solves the problem by time $\tau(\xi)$ (according to the external clock), then the *task-oriented work complexity* W of algorithm Λ is:

$$W = W_{\mathcal{A}}(n, p, f) = \max_{\xi \in \mathcal{E}(\Lambda, \mathcal{A}),\ \|\xi|_{\mathcal{A}}\| \leq f} \left\{ \sum_{i=1}^{\tau(\xi)} W_i(\xi) \right\}.$$

Note that in Definition 3.6 the idling processors are not charged for work (since we count only task-oriented units of work).

Observe from the above definitions that the *work* measure is more "conservative" than the *task-oriented work* measure. Given an algorithm Λ that solves *Do-All* under adversary $\mathcal{A}$ then $W_{\mathcal{A}}(n, p, f) = O(S_{\mathcal{A}}(n, p, f))$, since $S_{\mathcal{A}}(n, p, f)$ counts the idle/wait steps, which are not included in $W_{\mathcal{A}}(n, p, f)$. The equality $W_{\mathcal{A}}(n, p, f) = S_{\mathcal{A}}(n, p, f)$ can be achieved, for example, by algorithms that perform at least one task during any fixed time period. Also note that Definitions 3.5 and 3.6 do not depend on the specifics of the target model of computation, e.g., whether it is message-passing or shared-memory. When presenting algorithmic solutions or lower/upper bounds, we explicitly state which work measure is assumed.

Message complexity. The efficiency of message-passing algorithms is additionally characterized in terms of their *message complexity*. Let Λ be an algorithm that solves a problem of size n with p processors under adversary $\mathcal{A}$. For an execution $\xi \in \mathcal{E}(\Lambda, \mathcal{A})$ denote by $M_i(\xi)$ the number of point-to-point messages sent at time i of the execution, according to some external clock not available to the processors (for synchronous computations, the external clock is assumed to run in synchrony with the processors' local clocks).

Definition 3.7 (message complexity) Let Λ be an algorithm that solves a problem of size n with p processors under adversary $\mathcal{A}$. If execution $\xi \in \mathcal{E}(\Lambda, \mathcal{A})$, where $\|\xi|_{\mathcal{A}}\| \leq f$, solves the problem by time $\tau(\xi)$ (according to the external clock), then the *message complexity* M of algorithm Λ is:

$$M = M_{\mathcal{A}}(n, p, f) = \max_{\xi \in \mathcal{E}(\Lambda, \mathcal{A}),\ \|\xi|_{\mathcal{A}}\| \leq f} \left\{ \sum_{i=1}^{\tau(\xi)} M_i(\xi) \right\}.$$

Note that when processors communicate using broadcasts or multicasts, each broacast / multicast is counted as the number of point-to-point messages from the sender to each receiver.

Read and write memory access concurrency. In *synchronous shared-memory* systems, we are also interested in studying the read and write memory access concurrency of *Write-All* algorithms. Consider a step of a synchronous parallel computation, where a particular location is written by $x \leq p$ processors. Then $x - 1$ of these writes are potentially "redundant", because a single write suffices. The following read and write concurrency measures, introduced by Kanellakis, Michailidis, and Shvartsman in [66], assess the worst case number of redundant read and write memory accesses.

Definition 3.8 (read and write concurrency) Let Λ be a synchronous shared-memory algorithm that solves a problem of size n with p processors under adversary $\mathcal{A}$. Consider an execution $\xi \in \mathcal{E}(\Lambda, \mathcal{A})$ with $\|\xi|_{\mathcal{A}}\| \leq f$ that solves the problem by time $\tau(\xi)$. If at time i $(1 \leq i \leq \tau(\xi))$, $p_i^{\mathrm{r}}(\xi)$ processors complete reads from $n_i^{\mathrm{r}}(\xi)$ distinct shared memory locations and $p_i^{\mathrm{w}}(\xi)$ processors complete writes to $n_i^{\mathrm{w}}(\xi)$ distinct locations, then we define:

(i) the read concurrency ρ of Λ as:

$$\rho = \rho_{\mathcal{A}}(n, p, f) = \max_{\xi \in \mathcal{E}(\Lambda, \mathcal{A}),\ \|\xi|_{\mathcal{A}}\| \leq f} \left\{ \sum_{i=1}^{\tau(\xi)} \big(p_i^{\mathrm{r}}(\xi) - n_i^{\mathrm{r}}(\xi)\big) \right\},$$

(ii) the write concurrency ω of Λ as:

$$\omega = \omega_{\mathcal{A}}(n, p, f) = \max_{\xi \in \mathcal{E}(\Lambda, \mathcal{A}),\ \|\xi|_{\mathcal{A}}\| \leq f} \left\{ \sum_{i=1}^{\tau(\xi)} \big(p_i^{\mathrm{w}}(\xi) - n_i^{\mathrm{w}}(\xi)\big) \right\}.$$

Chapter 4

Perfect Knowledge: Do-All with Crashes

In this chapter we consider *synchronous* crash-prone processors under the assumption of perfect knowledge, where an oracle provides termination and load-balancing information to the processors (see paragraph "The assumption of perfect knowledge" in Section 3.1). The assumption of perfect knowledge abstracts away communication and scheduling issues and allows us to focus on the effects of processor failures on the efficiency of *Do-All*. We present a *complete* analysis of *Do-All*$^{\mathcal{O}}_{\mathcal{A}_S}(n,p,f)$ and r-*Do-All*$^{\mathcal{O}}_{\mathcal{A}_S}(n,p,f)$ work complexity that demonstrates precisely how failures affect efficiency. In particular, we provide matching upper and lower failure-sensitive bounds on work that are given as functions of n, p *and* f, the number of processor crashes, for the entire range of f. This also establishes the first non-trivial lower bound for *Do-All* for moderate number of failures ($f \leq p/\log p$). In later sections, we demonstrate the utility and generality of the results we obtain under the assumption of perfect knowledge by improving the analysis of three efficient algorithms: Algorithm AN [17] that solves *Do-All* in the message-passing model assuming reliable multicast (see Section 5.1), algorithm W [67], the best known algorithm that solves *Write-All* in the shared-memory model (see Section 6.1), and algorithm KMS [66] that solves *Write-All* with controlled memory access concurrency (see Section 6.2). By iteratively using these algorithms we also give improved failure-sensitive upper bounds for *iterative Do-All* in the corresponding models. Finally, our results under the perfect knowledge assumption yield insight about the bounds on task execution redundancy incurred when a central authority repeatedly allocates tasks to crash-prone processors (see Section 4.3).

4.1 Do-All Upper Bounds with Perfect Knowledge

To study the upper bounds for *Do-All* we give an oracle-based algorithm in Figure 1. The algorithm uses oracle $\mathcal{O}$ that performs the termination and load-balancing computation on behalf of the processors. In particular, during each synchronous iteration of an execution of the algorithm, the oracle $\mathcal{O}$ makes available to each processor i two values: *Oracle-complete*, a Boolean which takes the value true if and only if all tasks are complete at the beginning of this iteration, and *Oracle-task*(i), a natural number from $[n]$, whose value is a task identifier. *Oracle-task* is a function from processor identifiers to task identifiers, with the property that processors are only allocated to undone tasks, and that all such tasks are allocated a balanced number of processors. For example, if processors $i_1, \ldots, i_k \in [p]$ are alive and tasks $j_1, \ldots, j_\ell \in [n]$ are undone at the beginning of a given iteration of the algorithm, then *Oracle-task*$(i_s) = j_t$, where $t = (s - 1 \bmod \ell) + 1$.

```
for each processor PID = 1..p begin
    while not Oracle-complete
        perform task with TID = Oracle-task(PID)
end
```

Figure 1: Oracle-based algorithm.

We begin with a result shown by Kanellakis and Shvartsman [68]. This result was originally shown for the *Write-All* problem with memory snapshots (processors can access the entire shared-memory in constant time). It is not difficult to see that this result is trivially applicable to the *Do-All* problem with perfect knowledge (this is discussed in Section 2.3).

Lemma 4.1 [68] The *Do-All*$_{\mathcal{A}_S}^{\mathcal{O}}(n,p,f)$ problem can be solved with $f < p$ using work

$$S = O\left(n + p\frac{\log p}{\log\log p}\right).$$

Note that Lemma 4.1 does not show how, if at all, work depends on f. We now present an upper bound considering moderate number of crashes ($f \le p/\log p$).

Lemma 4.2 The *Do-All*$_{\mathcal{A}_S}^{\mathcal{O}}(n,p,f)$ problem can be solved with $f \le p/\log p$ using work

$$S = O\left(n + p\log_{\frac{p}{f}} p\right).$$

Proof: For an iteration of the algorithm in Figure 1, let Δf denote the number of processor crashes in this iteration. (Δf can be different for each iteration, though the sum of these for all iterations cannot exceed f.) We set $b = b(p,f) = \frac{p}{2f}$, and we define $S(n,p,f)$ to be the work required to solve *Do-All*$_{\mathcal{A}_S}^{\mathcal{O}}(n,p,f)$. Our goal is to show that for all u, p and f, the work $S(u,p,f)$ is no more than $16p + u + p\log_{\frac{p}{2f}}(\min(u,p))$, where $u \le n$ denotes the number of undone tasks. The proof proceeds by induction on u.

Base Case: Observe that when $u \le 16$, $S(u,p,f) \le 16p < 16p + u + p\log_b(\min(u,p))$, for all p and f.

Inductive Hypothesis: Assume that we have proved the theorem for all $u < \hat{u}$ ($\hat{u} \le n$) and all p and f.

Inductive Step: Consider $u = \hat{u}$. We investigate two cases:

Case 1: $p \leq \hat{u}$ (in particular, $\min(\hat{u}, p) = p$). In this case each processor is assigned to a unique task, hence

$$S(\hat{u}, p, f) \leq p + \max_{0 \leq \Delta f \leq f} S(\hat{u} - p + \Delta f, p - \Delta f, f - \Delta f).$$

As $p - \Delta f > 0$, $\hat{u} - p + \Delta f < \hat{u}$ and, by the induction hypothesis,

$$S(\hat{u}, p, f) \leq p + \max_{0 \leq \Delta f \leq f} \Big[16(p - \Delta f) + (\hat{u} - p + \Delta f) \\ + (p - \Delta f) \log_{b(p - \Delta f, f - \Delta f)} (\min(\hat{u} - p + \Delta f, p - \Delta f) \Big]$$

Now, $b(p - \Delta f, f - \Delta f) \geq b(p, f)$, and

$$\log_{b(p,f)} (\min(\hat{u} - p + \Delta f, p - \Delta f) \leq \log_{b(p,f)} (p - \Delta f),$$

so that

$$S(\hat{u}, p, f) \leq 16p + \hat{u} + p \log_{b(p,f)} p = 16p + \hat{u} + p \log_{b(p,f)} (\min(\hat{u}, p)),$$

as desired.

Case 2: $p > \hat{u}$ (in particular, $\min(\hat{u}, p) = \hat{u}$). In this case, by assumption we have

$$S(\hat{u}, p, f) \leq p + \max_{0 \leq \Delta f \leq f} S(\gamma \hat{u}, p - \Delta f, f - \Delta f),$$

where $\gamma = \gamma(\hat{u}, p, \Delta f)$ is the ratio of the number of the remaining tasks to $\hat{u}$ ($0 \leq \gamma < 1$).

Let $\phi = \Delta f / p \leq f/p < 1$, the fraction of processors which fail during this iteration; then $\phi/2 < \gamma < 2\phi$. $\Big($To see this, observe that

$$\frac{\phi p}{\lceil p/\hat{u} \rceil \hat{u}} = \frac{\phi p / \lceil p/\hat{u} \rceil}{\hat{u}} \leq \gamma \leq \frac{\phi p / \lfloor p/\hat{u} \rfloor}{\hat{u}} = \frac{\phi p}{\lfloor p/\hat{u} \rfloor \hat{u}}.$$

Let $p = c\hat{u}$, $c > 1$. Then

$$\frac{c}{\lceil c \rceil} \phi = \frac{\phi c \hat{u}}{\lceil c \rceil \hat{u}} \leq \gamma \leq \frac{\phi c \hat{u}}{\lfloor c \rfloor \hat{u}} = \frac{c}{\lfloor c \rfloor} \phi.$$

Now observe that $1 \leq \frac{c}{\lfloor c \rfloor} < 2$ and $1/2 < \frac{c}{\lceil c \rceil} \leq 1$, $\forall c > 1$, and hence, $\phi/2 < \gamma < 2\phi$, as desired.$\Big)$ Then,

$$S(\hat{u}, p, f) \leq p + \max_{\phi \in [0, f/p]} S(\gamma \hat{u}, (1 - \phi)p, f - \phi p).$$

As $\gamma\hat{u} < \hat{u}$, we may apply the induction hypothesis:

$$S(\hat{u}, p, f) \leq p + \max_{\phi \in [0, f/p]} \Big[16(1-\phi)p + \gamma\hat{u} + (1-\phi)p \log_{b'}(\min(\gamma\hat{u}, (1-\phi)p)) \Big],$$

where $b' = b(p - \phi p, f - \phi p)$. As above, $b' \geq b(p, f)$ and $\min(\gamma\hat{u}, (1-\phi)p)) \leq \gamma\hat{u}$, so that

$$S(\hat{u}, p, f) \leq p + \max_{\phi \in [0, f/p]} \Big[16(1-\phi)p + \gamma\hat{u} + (1-\phi)p \log_{b(p,f)}(\gamma\hat{u}) \Big].$$

To complete the proof, it suffices to show that for all $\phi \in [0, f/p]$,

$$15p + p\log_{b(p,f)} \hat{u} - (1-\phi)p\log_{b(p,f)}(\gamma\hat{u}) \geq 16(1-\phi)p - \hat{u}(1-\gamma).$$

Upper bounding $16(1-\phi)p - \hat{u}(1-\gamma)$ with $16(1-\phi)p$ and dividing through by p, it is sufficient to show that

$$15 + \log_{b(p,f)} \hat{u} - (1-\phi)\log_{b(p,f)}(\gamma\hat{u}) \geq 16(1-\phi),$$

or, equivalently,

$$\log_{b(p,f)} \hat{u} - (1-\phi)\log_{b(p,f)}(\gamma\hat{u}) \geq 1 - 16\phi.$$

We now focus on the left hand side of the above equation:

$$\log_{b(p,f)} \hat{u} - (1-\phi)\left[\log_{b(p,f)} \gamma + \log_{b(p,f)} \hat{u}\right] = \phi \log_{b(p,f)} \hat{u} + (1-\phi)\log_{b(p,f)} \gamma^{-1}.$$

Since $f \leq \frac{p}{\log(\min(\hat{u}, p))} = \frac{p}{\log \hat{u}}$, for any $\hat{u} > 16$ we have that $\frac{p}{2f} > 2$. Observe that,

$$\phi \log_{b(p,f)} \hat{u} + (1-\phi)\log_{b(p,f)} \gamma^{-1} \geq (1-\phi)\log_{b(p,f)} \gamma^{-1}$$

since $\hat{u} \geq p/f > p/2f$. (Note that if $\hat{u} < p/f$, then all tasks are completed in this iteration.)

Recall that $\gamma^{-1} \geq (2\phi)^{-1}$ and $\phi < f/p$. Therefore,

$$(1-\phi)\log_{b(p,f)} \gamma^{-1} \geq (1-\phi)\log_{b(p,f)}(2\phi)^{-1} \geq 1 - 16\phi.$$

Evidently,

$$S = O\left(n + p + p\log_{\frac{p}{f}}(\min(n, p)\right) = O\left(n + p\log_{\frac{p}{f}} p\right),$$

as desired. □

We now give our failure-sensitive upper-bound result.

Theorem 4.3 The *Do-All*$^{\mathcal{O}}_{\mathcal{A}_S}(n,p,f)$ problem can be solved using work

$$S = O\left(n + p\frac{\log p}{\log(p/f)}\right) \text{ when } f \le \frac{p}{\log p}, \text{ and}$$

$$S = O\left(n + p\frac{\log p}{\log\log p}\right) \text{ when } \frac{p}{\log p} < f < p.$$

Proof: This follows from Lemmas 4.1 and 4.2. $\square$

4.2 Do-All Lower Bounds

We now develop the lower bounds for *Do-All*$^{\mathcal{O}}_{\mathcal{A}_S}(n,p,f)$; these bounds match the upper bounds presented in Section 4.1. Note that the results in this section hold also for the *Do-All*$_{\mathcal{A}_S}(n,p,f)$ problem (without the oracle).

The following mathematical facts (from [67]) are used in the proofs.

Fact 4.1 If $a_1, a_2, \ldots, a_m$ $(m > 1)$ is a sorted list of nonnegative integers, then for all j $(1 \le j < m)$ we have $\left(1 - \frac{j}{m}\right)\sum_{i=1}^{m} a_i \le \sum_{i=j+1}^{m} a_i$.

Fact 4.2 Given $n \in \mathbb{N}$, $\kappa \in \mathbb{R}$, such that $n \cdot \kappa > 1$, $\kappa \le \frac{1}{2}$, and $\sigma \in \mathbb{N}$ such that $\sigma < \frac{\log n}{\log(\kappa^{-1})} - 1$, then the following inequality holds: $\lfloor \ldots \lfloor \lfloor n \underbrace{\cdot\, \kappa \rfloor \cdot \kappa \rfloor \ldots \cdot \kappa \rfloor}_{\sigma\ times} > 0$.

Proof: To show the result it suffices to show that, after dropping one floor and strengthening the inequality: $(\lfloor \ldots \lfloor \lfloor n \underbrace{\cdot\, \kappa \rfloor \cdot \kappa \rfloor \ldots \cdot \kappa \rfloor}_{\sigma - 1\ times} \cdot\, \kappa) - 1 > 0$, or that $\lfloor \ldots \lfloor \lfloor n \underbrace{\cdot\, \kappa \rfloor \cdot \kappa \rfloor \ldots \cdot \kappa \rfloor}_{\sigma - 1\ times} > \frac{1}{\kappa}$.

Applying this transformation for $\sigma - 1$ more steps, we see that it suffices to show that $n > \frac{1}{\kappa^{\sigma}} + \frac{1}{\kappa^{\sigma-1}} + \ldots + \frac{1}{\kappa}$, or, using geometric progression summation, that $n > \frac{(\kappa^{-1})^{\sigma+1} - (\kappa^{-1})}{(\kappa^{-1}) - 1}$.

We observe that $$(\kappa^{-1})^{\sigma+1} > \frac{(\kappa^{-1})^{\sigma+1} - (\kappa^{-1})}{(\kappa^{-1}) - 1}$$

for $\kappa \leq \frac{1}{2}$, thus it is enough to show that $n > (\kappa^{-1})^{\sigma+1}$. After taking logarithms of both sides of the inequality, $\log n > (\sigma+1)\log(\kappa^{-1})$, and so it suffices to have $\sigma < \frac{\log n}{\log(\kappa^{-1})} - 1$. □

We now define a specific adversarial strategy of adversary $\mathcal{A}_S$ used to derive our lower bounds. Let Λ be an iterative algorithm that solves the *Do-All* problem. Let p_i be the number of processors remaining at the end of the i^{th} iteration of an execution of Λ and let u_i denote the number of tasks that remain to be done at the end of iteration i. Initially, $p_0 = p$ and $u_0 = n$. The adversarial strategy is defined assuming the same initial number of tasks and processors, that is, $p_0 = n_0$. The strategy of the adversary is defined for each iteration of the algorithm. Based on a variable κ, defined in the interval $(0, 1/2)$, the adversary determines which processors will be allowed to work and which will be stopped in a given iteration. We call this adversarial strategy $\mathfrak{A}$.

Adversarial strategy $\mathfrak{A}$:

Iteration 1**:** The adversary chooses $u_1 = \lfloor \kappa u_0 \rfloor$ tasks with the least number of processors assigned to them. This can be done since the adversary is omniscient; it knows all the actions to be performed by Λ (as well as any advice provided by the oracle). The adversary then crashes the processors assigned to these tasks, if any.

Iteration i**:** Among u_{i-1} tasks remaining after the iteration $i-1$, the adversary chooses $u_i = \lfloor \kappa u_{i-1} \rfloor$ tasks with the least number of processors assigned to them and crashes these processors.

*Termination***:** The adversary continues for as long as $u_i > 1$. As soon as $u_i = 1$, the adversary allows all remaining processors to perform the single remaining task, and Λ terminates.

We now study the adversarial strategy $\mathfrak{A}$ and derive lower bound results.

Remark 4.1 Relationship between n and κ: If κ is chosen so that $\kappa \cdot n \leq 1$ then by the adversarial strategy $\mathfrak{A}$, an algorithm solving *Do-All* may be able to solve it in a constant number of iterations (namely two) with work $O(p)$. This is because $u_1 = \lfloor \kappa u_0 \rfloor \leq \kappa n \leq 1$. Henceforth we consider κ to be such that $\kappa \cdot n > 1$.

Lemma 4.4 For adversarial strategy $\mathfrak{A}$, if at iteration i the number of remaining tasks is $u_{i-1} > 1$, then

(a) $u_i = \lfloor \ldots \lfloor \lfloor n \underbrace{\cdot \, \kappa \rfloor \cdot \kappa \rfloor \ldots \cdot \kappa}_{i \; times} \rfloor$, and

(b) $p_i \geq (1-\kappa)^i \, p_0$.

Proof: Part (a) is immediate from the definition of $\mathfrak{A}$. To express the number of surviving processors p_i for part (b), we use Fact 4.1 with the following definitions:

Let $m = u_{i-1}$, and let $a_1, \ldots, a_m$ be the quantities of processors assigned to each task, sorted in ascending order. Let a_m also include the quantity of any un-assigned processors, i.e., a_1 is the least number of processors assigned to a task, a_2 is the next least quantity of processors, etc. (In other words, $a_1 \leq a_2 \leq \ldots \leq a_m$.) Let $j = u_i$. Thus the adversary stops exactly $\sum_{i=1}^{j} a_i$ processors. At the beginning of iteration i, the number of processors $p_{i-1} = \sum_{i=1}^{m} a_i$, therefore, the number of surviving processors $p_i = \sum_{i=j+1}^{m} a_i$.
Using Fact 4.1, we have $p_i \geq (1 - \dfrac{u_i}{u_{i-1}}) p_{i-1}$, and after substituting for $u_i = \lfloor \kappa u_{i-1} \rfloor$ we have

$$p_i \geq \left(1 - \frac{\lfloor \kappa u_{i-1} \rfloor}{u_{i-1}}\right) p_{i-1} \geq (1-\kappa)\, p_{i-1} \geq (1-\kappa)^i \, p_0,$$

as desired. $\square$

Lemma 4.5 Given any algorithm solving the *Do-All*$^{\mathcal{O}}_{\mathcal{A}_S}(p, p, f)$ problem ($p = n$), the adversarial strategy $\mathfrak{A}$ will cause the algorithm to cycle through at least $\frac{\log p}{\log(\kappa^{-1})} - 1$ iterations.

Proof: Let τ be the earliest iteration when the last task is performed. We use Fact 4.2 with σ the largest integer such that $\sigma < \log p / \log(\kappa^{-1}) - 1$. Then $u_\sigma = \lfloor \ldots \lfloor \lfloor p \underbrace{\cdot \kappa \rfloor \cdot \kappa \rfloor \ldots \cdot \kappa \rfloor}_{\sigma\ times} > 0$, and so τ must be greater than σ because $u_\tau = 0$. Thus, $\tau \geq \dfrac{\log p}{\log(\kappa^{-1})} - 1 > \sigma$. □

Lemma 4.6 Given any algorithm Λ that solves the *Do-All*$^{\mathcal{O}}_{\mathcal{A}_S}(p, p, f)$ problem ($p = n$) with $f < p$, the adversarial strategy $\mathfrak{A}$ with $\kappa = \frac{1}{\log p}$ causes work $S = \Omega\left(p \dfrac{\log p}{\log \log p}\right)$.

Proof: We first assume that $p > 4$ (we aim to establish an asymptotic result, and this eliminates uninteresting cases). Since $\kappa = 1/\log p$, we have that $\kappa \in (0, 1/2)$ when $p > 4$. From Lemma 4.4(a) and Lemma 4.5 we see that $\mathfrak{A}$ will cause algorithm Λ to iterate at least $\tau = (\log p / \log \log p) - 1$ times. Now observe that the work must be at least $p_\tau \cdot \tau$, where p_τ is the number of surviving processors after Λ terminates. From Lemma 4.4(b) we have that $p_\tau \geq (1 - \kappa)^\tau p_0 = (1 - \frac{1}{\log p})^\tau p$. Therefore,

$$
\begin{aligned}
p_\tau \;&\geq\; p\left(1 - \tfrac{1}{\log p}\right)^{\frac{\log p}{\log \log p} - 1} &&\geq\; p\left(1 - \tfrac{1}{\log p}\right)^{\frac{\log p}{\log \log p}} \\
&\geq\; p\left(1 - \left(\tfrac{1}{\log p}\right) \cdot \left(\tfrac{\log p}{\log \log p}\right)\right) &&=\; p - \tfrac{p}{\log \log p}.
\end{aligned}
$$

Let f_τ denote the actual number of crashes caused by the adversary. Then, $f_\tau = p - p_\tau \leq p - p + \frac{p}{\log \log p} = \frac{p}{\log \log p} < p$. Hence, $\mathfrak{A}$ when using this specific κ does not exceed the allowed number of crashes. Now, the work caused by $\mathfrak{A}$ is:

$$
S = \Omega(p_\tau \cdot \tau) = \Omega\left(\left(p - \frac{p}{\log \log p}\right) \cdot \left(\frac{\log p}{\log \log p} - 1\right)\right) = \Omega\left(p \frac{\log p}{\log \log p}\right).
$$

This completes the proof. □

Corollary 4.7 Given any algorithm Λ that solves the *Do-All*$^{\mathcal{O}}_{\mathcal{A}_S}(n, p, f)$ problem ($p \leq n$) there exists an adversarial strategy that causes work $S = \Omega\left(n + p \dfrac{\log p}{\log \log p}\right)$.

Proof: Note that $S = \Omega(n)$ because all tasks must be performed. From Lemma 4.6 we know that *Do-All*$^{\mathcal{O}}_{\mathcal{A}_S}(p, p, f)$ requires $\Omega(p \log p / \log \log p)$ work. Given that work is nondecreasing in n (as follows from Definition 3.5) we obtain the desired result by combining the two bounds. $\square$

Observe that Lemma 4.6 and Corollary 4.7, by themselves, do not show how work depends on f. We now give lower bounds considering moderate number of crashes ($f \leq p/\log p$).

Lemma 4.8 Given any algorithm Λ that solves the *Do-All*$^{\mathcal{O}}_{\mathcal{A}_S}(p, p, f)$ problem ($p = n$), the adversarial strategy $\mathfrak{A}$ with $(\kappa^{-1})\log(\kappa^{-1}) = \frac{p \log p}{f}$ and $f \leq \frac{p}{\log p}$ causes work $S = \Omega\left(p \log_{\frac{p}{f}} p\right)$.

Proof: We assume that $p > 4$ (we aim to establish an asymptotic result, and this eliminates uninteresting cases). From $(\kappa^{-1})\log(\kappa^{-1}) = \frac{p \log p}{f}$, $f \leq \frac{p}{\log p}$, and $p > 4$ we see that $\log(\kappa^{-1}) > 4\kappa$. This implies that $\kappa \in (0, 1/2)$. Hence, from Lemma 4.5 we have that $\mathfrak{A}$ will cause algorithm Λ to iterate at least $\tau = (\log p / \log(\kappa^{-1})) - 1$ times.

Now observe that the work must be at least $p_\tau \cdot \tau$, where p_τ is the number of surviving processors after Λ terminates. Recall from Lemma 4.4(b) that $p_\tau \geq (1-\kappa)^\tau p_0$. Therefore,

$$
\begin{array}{rclcl}
p_\tau & \geq & p\,(1-\kappa)^\tau & \geq & p\,(1-\kappa)^{\frac{\log p}{\log(\kappa^{-1})}-1} \\
& \geq & p\,(1-\kappa)^{\frac{\log p}{\log(\kappa^{-1})}} & \geq & p\left(1 - \kappa \cdot \frac{\log p}{\log(\kappa^{-1})}\right) \\
& = & p\left(1 - \left(\frac{\kappa}{\log(\kappa^{-1})}\right)\log p\right) & = & p\left(1 - \left(\frac{f}{p \log p}\right)\log p\right) \\
& = & p - f. & &
\end{array}
$$

Let f_τ denote the actual number of crashes caused by the adversary. Then, $f_\tau = p - p_\tau \leq p - (p - f) = f$. Hence, $\mathfrak{A}$ when using this specific κ does not exceed the allowed number of crashes ($f \leq p/\log p$).

Recall that $(\kappa^{-1})\log(\kappa^{-1}) = \frac{p\log p}{f}$, therefore, $(\kappa^{-1}) = \Theta\left(\frac{\frac{p\log p}{f}}{\log(\frac{p\log p}{f})}\right)$. Thus,

$$\log(\kappa^{-1}) = \Theta\left(\log\left(\frac{p\log p}{f}\right) - \log\log\left(\frac{p\log p}{f}\right)\right) = \Theta\left(\log\left(\frac{p\log p}{f}\right)\right).$$

Then, noting that $p_\tau \geq p - f \geq p - p/\log p = \Theta(p)$ and that $\kappa \cdot p > 1$ (see Remark 4.1), we assess the work S caused by $\mathfrak{A}$ as follows:

$$S = \Omega(p_\tau \cdot \tau) = \Omega\left(p \cdot \frac{\log p}{\log(\kappa^{-1})}\right) = \Omega\left(p + p\frac{\log p}{\log(\frac{p\log p}{f})}\right).$$

Now recall that $p/f \geq \log p$. Hence, for any $p > 4$ we have that $p/f > 2$ and that $\log((p\log p)/f) = \log(p/f) + \log\log p = \Theta(\log(p/f))$. From the above,

$$S = \Omega\left(p + p\frac{\log p}{\log(\frac{p}{f})}\right) = \Omega\left(p\log_{\frac{p}{f}} p\right).$$

This completes the proof. □

Corollary 4.9 Given any algorithm Λ that solves the *Do-All*$^{\mathcal{O}}_{\mathcal{A}_S}(n,p,f)$ problem ($p \leq n$), there exists an adversarial strategy that causes $f \leq \frac{p}{\log p}$ crashes, and work $S = \Omega\left(n + p\log_{\frac{p}{f}} p\right)$.

Proof: Note that $S = \Omega(n)$ because all tasks must be performed. From Lemma 4.8 we know that *Do-All*$^{\mathcal{O}}_{\mathcal{A}_S}(p,p,f)$ requires $\Omega(p\log_{\frac{p}{f}} p)$ work, for $f \leq p/\log p$. Given that work is nondecreasing in n we obtain the desired result by combining the two bounds. □

We now give our failure-sensitive lower-bound result.

Theorem 4.10 Given any algorithm Λ that solves the *Do-All*$^{\mathcal{O}}_{\mathcal{A}_S}(n,p,f)$ problem there exists an adversarial strategy that causes work

$$S = \Omega\left(n + p\frac{\log p}{\log(p/f)}\right) \text{ when } f \leq \frac{p}{\log p}, \text{ and}$$

$$S = \Omega\left(n + p\frac{\log p}{\log\log p}\right) \text{ when } \frac{p}{\log p} < f < p.$$

Proof: For the range of failures $f \leq p/\log p$, per Corollary 4.9, the work is $\Omega(n+p\log_{p/f} p)$.

From Corollary 4.9 we also obtain the fact that when $f = p/\log p$ then work must be $\Omega\,(n + p\log p/\log\log p)$. Note that this is the worst case work for any f (see Corollary 4.7). Therefore, for the range $p/\log p < f < p$, the adversary establishes this worst case work using the initial $p/\log p$ failures. □

4.3 Iterative Do-All

Do-All algorithms have been used in developing simulations of failure-free algorithms on failure-prone processors. This is done by iteratively using a *Do-All* algorithm to simulate the steps of the failure-free processors. We study the *iterative Do-All* problems to understand the complexity implications of iterative use of *Do-All* algorithms.

In studying simulations, a *Do-All*$_{\mathcal{A}_S}(n,p,f)$ solution abstracts the setting where p physical crash-prone processors simulate n virtual processors, such that each task i among the n tasks in *Do-All* represents a single step of the virtual processor i. The *iterative Do-All* then models the simulation of multiple steps of the virtual processors.

In principle r-*Do-All*$_{\mathcal{A}_S}(n,p,f)$ can be solved by running an algorithm for *Do-All*$_{\mathcal{A}_S}(n,p,f)$ for r iterations. For example, r-*Do-All*$^{\mathcal{O}}_{\mathcal{A}_S}(n,p,f)$ can be solved by running the oracle-based algorithm in Figure 1 for r iterations. If the work of a *Do-All* solution is S, then the work of the r-*iterative Do-All* is at most $r \cdot S$. However we show that it is possible to obtain a finer result that takes into account the diminishing number of failures "available" to the adversary. We refer to each *Do-All* iteration as a *round* of r-*Do-All*$_{\mathcal{A}_S}(n,p,f)$.

For the model of perfect knowledge we obtain the following failure-sensitive upper bound on work.

Theorem 4.11 The r-*Do-All*$^{\mathcal{O}}_{\mathcal{A}_S}(n,p,f)$ problem can be solved using work

$$S = O\left(r\cdot\left(n + p\frac{\log p}{\log(pr/f)}\right)\right) \text{ when } f \le \frac{pr}{\log p}, \text{ and}$$

$$S = O\left(r\cdot\left(n + p\frac{\log p}{\log\log p}\right)\right) \text{ when } \frac{pr}{\log p} < f < p.$$

Proof: Let r_i denote the i^{th} round of the iterative *Do-All*. Let p_i be the number of active processors at the beginning of r_i and f_i be the number of crashes during r_i. Note that $p_1 = p$, where r_1 is the first round of r-*Do-All*$^{\mathcal{O}}_{\mathcal{A}_S}(n,p,f)$ and that $p_i \le p$. We consider two cases:

Case 1: $f > \frac{pr}{\log p}$. Consider a round r_i. From Theorem 4.3 we see that the work for this round is $O\left(n + p_i \log_{p_i/f_i} p_i\right)$ when $f_i \le p_i/\log p_i$ and $O\left(n + p_i \log p_i/\log\log p_i\right)$ otherwise. However in this case, we can have $f_i = \Theta\left(p/\log p\right)$ for all r_i without "running out" of processors. Thus,

$$S_1 = O\left(r\cdot\left(n + p\frac{\log p}{\log\log p}\right)\right).$$

Case 2: $f \le \frac{pr}{\log p}$. First observe that any reasonable adversarial strategy would not kill more that $p_i/\log p_i$ processors in round r_i, since it would not cause more work than $O(n + p_i \log p_i/\log\log p_i)$ (which is achieved when $f_i \ge p_i/\log p_i$). Therefore, we consider $f_i \le p_i/\log p_i$ for all rounds r_i. Hence, the work in every round r_i (per Theorem 4.3) is $O\left(n + p_i \log p_i/\log(p_i/f_i)\right) = O\left(n + p\log p/\log(p/f_i)\right)$.

Let $S(n,p,f)$ be this one-round upper bound. As $f = \sum f_i$, an upper bound on r-*Do-All*$^{\mathcal{O}}_{\mathcal{A}_S}(n,p,f)$ can be given by maximizing $\sum_i S(n,p_i,f_i)$ over all such adversarial patterns. As $S(\cdot,\cdot,\cdot)$ is monotone in p, we may assume that $p_i = p$ for the purposes of the upper bound. We show that this maximum is attained at $f_1 = f_2 = \ldots = f_r$. For simplicity, treat f_i as a continuous parameter and consider the factor in the single round work expression (given above) that depends on f_i : $c/\log(\frac{p}{f_i})$, where c is the constant hidden by the $O(\cdot)$ notation.

The first derivative over f_i is $\dfrac{\partial}{\partial f_i}\left(c/\log\left(\dfrac{p}{f_i}\right)\right) = c/f_i(\log p - \log f_i)^2$, and its second derivative is $\frac{\partial^2}{\partial f_i^2}\left(c/\log\left(\frac{p}{f_i}\right)\right) = 2c/f_i^2(\log p - \log f_i)^3 - c/f_i^2(\log p - \log f_i)^2$. Observe that the second derivative is negative in the domain considered (assuming $p > 16$). Hence the first derivative is decreasing (with f_i). In this case, given any two $f_i,\ f_j$ where $f_i > f_j$, the adversarial pattern obtained by replacing f_i with $f_i - \epsilon$ and f_j by $f_j + \epsilon$ (where $\epsilon < (f_i - f_j)/2$) results in increased work. This implies that the sum maximized when all f_is are equal, specifically when $f_i = f/r$.

As the above upper bound on the sum $\sum_i S(n, p_i, f_i)$ is valid over *all* f_i in this range, it holds in particular for the choices made by the adversary which must, of course, cause an integer number of faults in each round. Therefore,

$$S_2 = O\left(r \cdot \left(n + p\frac{\log p}{\log(\frac{pr}{f})}\right)\right).$$

The result then follows by combining the two cases. □

We now show a matching lower bound.

Theorem 4.12 Given any algorithm that solves the r-*Do-All*$^{\mathcal{O}}_{\mathcal{A}_S}(n, p, f)$ problem, there exists an adversarial strategy that causes work

$$S = \Omega\left(r \cdot \left(n + p\frac{\log p}{\log(pr/f)}\right)\right) \text{ when } f \leq \frac{pr}{\log p}, \text{ and}$$

$$S = \Omega\left(r \cdot \left(n + p\frac{\log p}{\log\log p}\right)\right) \text{ when } \frac{pr}{\log p} < f < p.$$

Proof: Consider two cases:

Case 1: $f > \frac{pr}{\log p}$. In this case the adversary may crash $p/\log p$ processors in every round of r-*Do-All*$^{\mathcal{O}}_{\mathcal{A}_S}(n, p, f)$. Note that for this adversary $\Omega(p)$ processors remain alive during the first $\lceil r/2 \rceil$ rounds. Per Theorem 4.10 this results in $\lceil r/2 \rceil \cdot \Omega\left(n + p\log p/\log\log p\right) = \Omega\left(Nr + pr\log p/\log\log p\right)$ work.

Case 2: $f \leq \frac{pr}{\log p}$. In this case the adversary ideally would crash f/r processors in every round. It can do that in the case where r divides f. If this is not the case, then the adversary crashes $\lceil f/r \rceil$ processors in r_A rounds and $\lfloor f/r \rfloor$ in r_B rounds in such a way that $r = r_A + r_B$. Again considering the first half of the rounds and appealing to Theorem 4.10 results in a $\Omega\left(Nr + pr \log_{pr/f} p\right)$ lower bound for work. Note that we consider only the case where $r \leq f$; otherwise the work is trivially $\Omega(rN)$.

The result then follows by combining the two cases. □

Application of iterative Do-All: The bounds we obtained for *Do-All* and *iterative Do-All* under the assumption of perfect knowledge, yield insight about the bounds on task execution redundancy in settings where a server repeatedly allocates task to processors (e.g., SETI [74]).

Consider the setting where a central server repeatedly allocates tasks to crash-prone processors. When a processor completes a task, it reports this to the server. If a server detects processor failures, it must re-allocate the tasks to other processors. Processor crashes might cause some tasks to be executed more than once. Our results obtained for synchronous *Do-All*$^{\mathcal{O}}_{\mathcal{A}_S}(n, p, f)$ and r-*Do-All*$^{\mathcal{O}}_{\mathcal{A}_S}(n, p, f)$ are relevant to the bounds on task execution redundancy in such a setting. When the server allocates n similar, independent and idempotent tasks to p synchronous, crash-prone processors, then, per Theorems 4.3 and 4.10, the total number of task executions is $\Theta\left(n + p\frac{\log p}{\log(pr/f)}\right)$ when $f \leq \frac{p}{\log p}$, and $\Theta\left(n + p\frac{\log p}{\log\log p}\right)$ when $\frac{p}{\log p} < f < p$. Similarly, if the server allocates r "waves" of n tasks (so that a task-wave is completed before the next is begun) to p synchronous, crash-prone processors, then per Theorems 4.11 and 4.12, the total number of task executions is $\Theta\left(r \cdot \left(n + p\frac{\log p}{\log(pr/f)}\right)\right)$ when $f \leq \frac{pr}{\log p}$, and $\Theta\left(r \cdot \left(n + p\frac{\log p}{\log\log p}\right)\right)$ when $\frac{pr}{\log p} < f < p$.

Chapter 5

Message-Passing: Do-All with Crashes

We present failure-sensitive bounds on work and messages for the *Do-All*$_{\mathcal{A}_S}(n, p, f)$ problem with synchronous message-passing processors, for the entire range of f ($1 \leq f < p$). In Section 5.1 we assume that reliable multicast [60] is available (if a processor crashes while multicasting a message, then either all targeted processors receive the message or none do), whereas in Section 5.2 we assume that only traditional point-to-point messaging is available (multicast is not reliable).

5.1 Failure-Sensitive Bounds with Reliable Multicast

In this section we give a new, failure-sensitive, analysis of algorithm AN [17] and establish new complexity results for the iterative *Do-All* in the message-passing model. We achieve this by separately assessing the cost of tolerating failures and the cost of achieving perfect knowledge (that is, perfect load-balancing). The first analysis is derived from the results obtained under the assumption of perfect knowledge. The latter is derived from the structure of the algorithm.

Algorithm AN presented by Chlebus *et al.* [17] uses a multiple-coordinator approach to solve *Do-All*$_{\mathcal{A}_S}(n, p, f)$ on crash-prone synchronous message-passing processors ($p \leq n$). The model assumes that messages incur a known bounded delay and that reliable multicast [60] is available (when a processor multicasts a message to a collection of processors, either all messages are delivered to non-faulty processors or no messages are delivered).

5.1.1 Description of Algorithm AN

We now give a brief description of the algorithm, but to avoid a complete restatement, we refer the reader to [17]. Algorithm AN proceeds in a *loop* which is iterated until all the tasks are executed. A single iteration of the loop is called a *phase*. A phase consists of three consecutive *stages*. Each stage consists of three steps. In each stage processors use the first step to receive messages sent in the previous stage, the second step to perform local computation, and the third step to send messages. A processor can be a *coordinator* or a *worker*. A phase may have multiple coordinators. The number of processors that assume the coordinator role is determined by the *martingale principle*: if none of the expected coordinators survive through the entire phase, then the number of coordinators for the next phase is doubled. If at least one coordinator survives in a given phase, then in the next phase there is only one coordinator. A phase that is completed with at least one coordinator alive is called *attended*, otherwise it is called *unattended*.

Processors become coordinators and balance their loads according to each processor's *local view*. A local view contains the set of ids of the processors assumed to be alive. The local view is partitioned into *layers*. The first layer contains one processor id, the second two ids, the i^{th} contains 2^{i-1} ids.

Given a phase, in the first stage, the processors perform a task according to the load-balancing rule derived from their local views and report the completion of the task to the coordinators of that phase (determined by their local views). In the second stage, the coordinators gather the reports, they update the knowledge of the done tasks and they multicast this information to the processors that are assumed to be alive. In the last stage, the processors receive the information sent by the coordinators and update their knowledge of done tasks and their local views. Given the full details of the algorithm, it is not difficult to see that the combination of coordinators and local views allows the processors to obtain the information that would be available from the oracle $\mathcal{O}$ in the algorithm in Figure 1 of Section 4.1.

It is shown in [17] that the work of algorithm AN is $S = O((n + p\log p/\log\log p)\log f)$ and its message complexity is $M = O(n + p\log p/\log\log p + fp)$, for $p \le n$.

In the rest of this section we present the new analysis of work and message complexity of algorithm AN. Throughout we assume that the algorithm correctness is shown as in [17].

5.1.2 Analysis of Work Complexity

To assess the work S, we consider separately all the attended phases and all the unattended phases of the execution. Let S_a be the part of S spent during all the attended phases and S_u be the part of S spent during all the unattended phases. Hence we have $S = S_a + S_u$.

Lemma 5.1 [17] In any execution of algorithm AN with $f < p$ we have $S_a = O\left(n+p\frac{\log p}{\log\log p}\right)$ and $S_u = O\left(S_a \log f\right)$.

We now give the new analysis of algorithm AN.

Lemma 5.2 In any execution of algorithm AN with $f \le \frac{p}{\log p}$ we have $S_a = O\left(n + p\log_{\frac{p}{f}} p\right)$.

Proof: Given a phase i of an execution of algorithm AN, we define p_i to be the number of live processors and u_i to be the number of undone tasks at the beginning of the phase ($p_0 = p$ and $u_0 = n$). Let $\alpha_1, \alpha_2, \ldots \alpha_\tau$, denote all the attended phases of this execution (α_τ is the last phase of the execution).

Observe that for all $\alpha_i,\ 1 \leq i \leq \tau - 1$ it holds that (a) $u_{a_i} > u_{a_{i+1}}$, and (b) $p_{a_i} \geq p_{a_{i+1}}$. This follows from the construction of algorithm AN: Since phase α_i is attended, there is at least one coordinator, call it c, alive in phase α_i; c executes one task. Hence, at least one task is executed and consequently at least one task is removed from u_{a_i}. The number of processors can only decrease, since we do not allow restarts.

In [17], Section 3.2, it is shown that if at the beginning of phase a_i, the processors have consistent information on the number of surviving processors (p_{a_i}) and the number of remaining tasks (u_{a_i}), then the operational processors will have consistent information on $p_{a_{i+1}}$ and $u_{a_{i+1}}$ at the beginning of phase a_{i+1}. And since the processors have consistent information at a_0, that means that at the beginning of every attended phase, the surviving processors have consistent view of the system. Hence, the processors in attended phases can perform perfect load balancing, as in the case where the processors are assisted by the oracle $\mathcal{O}$, in the oracle model. Therefore, focusing only on the attended phases (and assuming that in the worst case no progress is made in unattended phases), we obtain the desired result by induction on the size of undone tasks u, as in the proof of Lemma 4.2. $\square$

Theorem 5.3 In any execution of algorithm AN we have work

$$S = O\left(\log f\left(n + p\frac{\log p}{\log(p/f)}\right)\right) \text{ when } f \leq \frac{p}{\log p}, \text{ and}$$

$$S = O\left(\log f\left(n + p\frac{\log p}{\log\log p}\right)\right) \text{ when } \frac{p}{\log p} < f < p.$$

Proof: This follows from Lemmas 5.1 and 5.2, and the fact that $S = S_a + S_u$. $\square$

5.1.3 Analysis of Message Complexity

To assess the message complexity M we consider separately all the attended phases and all the unattended phases of the execution. Let M_a be the number of messages sent during all the attended phases and M_u the number of messages sent during all the unattended phases. Hence we have $M = M_a + M_u$.

Lemma 5.4 [17] In any execution of algorithm AN with $f < p$ we have $M_a = O(S_a)$ and $M_u = O(fp)$.

Theorem 5.5 In any execution of algorithm AN we have

$$M = O\left(n + p\frac{\log p}{\log(p/f)} + fp\right) \text{ when } f \leq \frac{p}{\log p}, \text{ and}$$

$$M = O\left(n + p\frac{\log p}{\log\log p} + fp\right) \text{ when } \frac{p}{\log p} < f < p.$$

Proof: It follows from Lemmas 5.1, 5.2 and 5.4, and the fact that $M = M_a + M_u$. $\square$

5.1.4 Analysis of Message-Passing Iterative Do-All

We now consider the message-passing, synchronous r-*Do-All*$_{\mathcal{A}_S}(n,p,f)$ problem.

Theorem 5.6 The r-*Do-All*$_{\mathcal{A}_S}(n,p,f)$ problem can be solved on synchronous crash-prone message-passing processors when $f \leq \frac{pr}{\log p}$ with

$$S = O\left(r \cdot \log\left(\frac{f}{r}\right) \cdot \left(n + p\frac{\log p}{\log(pr/f)}\right)\right) \text{ and } M = O\left(r \cdot \left(n + p\frac{\log p}{\log(pr/f)}\right) + fp\right),$$

and when $\frac{pr}{\log p} < f < p$ with

$$S = O\left(r \cdot \log\left(\frac{f}{r}\right) \cdot \left(n + p\frac{\log p}{\log\log p}\right)\right) \text{ and } M = O\left(r \cdot \left(n + p\frac{\log p}{\log\log p}\right) + fp\right).$$

Proof: The iterative *Do-All* can be solved by running algorithm AN on r instances of size n in sequence. We call this algorithm AN*. To analyze the efficiency of AN* we use the same approach as in the proof of Theorem 4.11. In the current context we base our work complexity arguments on the result of Theorem 5.3, and we base our message complexity arguments on the result of Theorem 5.5. □

5.2 Failure-Sensitive Bounds without Reliable Multicast

In this section we present a new efficient synchronous message-passing algorithm for *Do-All*$_{\mathcal{A}_S}(n,p,f)$. The new algorithm has work complexity comparable to algorithm AN [17], however it uses simple point-to-point messaging. This algorithm achieves better work complexity than the algorithm of Galil *et al.* [44] (the previously best known algorithm not relying on reliable multicast) while obtaining the same asymptotic message complexity. The new algorithm does not use coordinator-based or checkpointing-based strategies to implement information sharing among processors (as the previously mentioned algorithms do). Instead, it uses an approach where processors share information using a gossip algorithm we developed to solve the gossip problem in synchronous message-passing systems with processor crashes. Our gossip algorithm achieves better message complexity than the previously best known algorithm of Chlebus and Kowalski [21], while obtaining the same asymptotic time complexity. The point-to-point messaging is constrained by means of a communication graph that represents a certain subset of the edges in a complete communication network. Processors send messages based on permutations with certain properties that we show to exist. We first define the gossip problem and relevant measures of efficiency (Section 5.2.1). We then present combinatorial tools that we use in the analysis of our gossip algorithm (Section 5.2.2). Then we present and analyze our gossip algorithm (Section 5.2.3). Finally we present and analyze our *Do-All* algorithm (Section 5.2.4).

5.2.1 The Gossip Problem

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

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

Definition 5.1 The *Gossip* problem: Given a set of p processors, where initially each processor has a distinct piece of information, called a *rumor*, the goal is for each processor to learn all the rumors in the presence of processor crashes. The following conditions must be satisfied:

(1) Correctness: (a) All non-faulty processors learn the rumors of all non-faulty processors, (b) For every failed processor v, non-faulty processor w either knows that v has failed, or w knows v's rumor.

(2) Termination: Every non-faulty processor terminates its protocol.

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

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

Definition 5.2 (time complexity) Let Λ be an algorithm that solves a problem with p processors under adversary $\mathcal{A}$. If execution $\xi \in \mathcal{E}(\Lambda, \mathcal{A})$, where $\|\xi|_{\mathcal{A}}\| \leq f$, solves the problem by time $\tau(\xi)$, then the *time complexity* T of algorithm Λ is:

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

The message complexity is defined as in Definition 3.7 where the size of the problem is p: it is measured as the total number of point-to-point messages sent by the processors until the problem is solved. As before, when a processor communicates using a multicast, its cost is the total number of point-to-point messages.

The previously best deterministic solution for the *Gossip* problem in the message passing model under adversary $\mathcal{A}_S$ is due to Chlebus and Kowalski [21]. Their algorithm has $T = O(\log^2 p)$ time complexity and $M = O(p^{1.77})$ message complexity. As we will see, our gossip algorithm substantially improves on the message complexity of their algorithm while obtaining the same asymptotic time complexity.

5.2.2 Combinatorial Tools

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

5.2.2.1 Communication Graphs

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

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

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

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

Chlebus *et al.* [19] formulated the notion of a "good" component G_{Q_i} of a subgraph G_{V_i} of graph G by setting $Q_i = P(V_i)$, where P is a function that satisfies a certain property called property $\mathcal{R}$:

Definition 5.3 ([19]) Graph G satisfies PROPERTY $\mathcal{R}(p, f)$ if there is a function P, which assigns subgraph $P(R) \subseteq G$ to each subgraph $R \subseteq G$ of size at least $p - f$, such that the following hold:

$\mathcal{R}.1:\ P(R) \subseteq R.$

$\mathcal{R}.2:\ |P(R)| \geq |R|/7.$

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

$\mathcal{R}.4:$ If $R_1 \subseteq R_2$ then $P(R_1) \subseteq P(R_2)$.

Let $L(p, \Delta_0)$ denote the family of constructive regular graphs of p nodes and degree Δ_0, that have good expansion properties. Such graphs were introduced by Lubotzky, Phillips and Sarnak [79]. These graphs are defined and can be constructed for each number p' of the form $q(q^2 - 1)/2$, where q is a prime integer congruent to 1 modulo 4. The node degree Δ_0 can be any number such that $\Delta_0 - 1$ is a prime congruent to 1 modulo 4 and a quadratic nonresidue modulo q. It follows, from the properties of the distribution of prime numbers (see e.g. [26]), that Δ_0 can be selected to be a constant independent of p and q such that $p' = q(q^2 - 1)/2 = \Theta(p)$. Since for each p there is a number $p' = \Theta(p)$, we let each processor simulate $O(1)$ nodes, and we henceforth assume that p is as required so that $L(p, \Delta_0)$ can be constructed. In [19] the authors extended the result of Upfal [109], who showed that there is a function P' such that if R is a subgraph of $L(p)$ of size at least $\frac{71}{72} \cdot p$ then subgraph $P'(R)$ of R has size at least $|R|/6$ and diameter at most $30 \log p$. (These constants in the case of linear-size subgraphs can be improved, see [5].) Let G^k be the k-th power of graph G, that is, $G^k = (V, E')$, where the edge $(u, v) \in E'$ if and only if there is a path between u and v in G of length at most k.

The authors in [19] proved the following lemma.

Lemma 5.7 ([19]) For every $f < p$ there exists a positive integer j such that graph $L(p)^j$ has PROPERTY $\mathcal{R}(p, f)$. Moreover, the maximum degree Δ of graph $L(p)^j$ is $O\big((\frac{p}{p-f})^{2\log_\gamma \Delta_0}\big)$, for some absolute constant γ, which for $\Delta_0 = 74$ could be taken equal to $\gamma = 27/5$.

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

Definition 5.4 Graph $G = (V, E)$ has the *Compact Chain Property* $CCP(p, f, \varepsilon)$, if:

I. The maximum degree of G is at most $\big(\frac{p}{p-f}\big)^{1+\varepsilon}$,

II. For a given sequence $V_1 \supseteq \ldots \supseteq V_k$ $(V = V_1)$, where $|V_k| \geq p - f$, there is a sequence $Q_1 \supseteq \ldots \supseteq Q_k$ such that for every $i = 1, \ldots, k$:

(a) $Q_i \subseteq V_i$,

(b) $|Q_i| \geq |V_i|/7$, and

(c) the diameter of G_{Q_i} is at most $31 \log p$.

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

Lemma 5.8 For $p{>}2$, every $f{<}p$, and constant $\varepsilon{>}0$, there is a graph G of $O(p)$ nodes satisfying property $CCP(p, f, \varepsilon)$.

Proof: Notice that for $p - f \leq \sqrt{p^\varepsilon}$, the complete graph K_p satisfies property $CCP(p, f, \varepsilon)$, for every constant $\varepsilon > 0$. The same holds if $p - f \geq p/4$, by applying Lemma 5.7 and setting

$Q_i = P(G_i)$ (in this case Δ is constant). For the remainder of the proof we assume that $\sqrt{p^{\varepsilon}} < p - f < p/4$.

Fix f and $\varepsilon > 0$. Our candidate for graph G is a graph $L(p, \Delta)$, where we take the smallest possible $\Delta \geq 9 + \big(\frac{p}{p-f}\big)^{1+\varepsilon}$. (By properties of graphs L, we can find $\Delta = O\big(1 + \big(\frac{p}{p-f}\big)^{1+\varepsilon}\big)$). Let $\lambda = 2\sqrt{\Delta - 1}$ be the bound for the absolute value of the second eigenvalue of graph $L(p, \Delta)$ (see [79]). Alon and Chung [4] showed that for every set $R \subseteq V$, the number of edges in the subgraph induced by R (denoted by $e(R)$) can be bounded as follows:

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

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

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

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

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

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

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

Next we show that for every $S' \subseteq S$ of size at most $|S|/2$, we have $|N_S(S')| \geq 4|S'|/3$. By definition of S, the total number of edges incident to nodes in S' is at least $|S'|\Delta\big(1 - \frac{|R|}{2p}\big)$. On the other hand, using inequality (1) we obtain

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

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

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

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

We now show how to construct a sequence $Q_1 \supseteq \ldots \supseteq Q_k$ having a sequence $V_1 \supseteq \ldots \supseteq V_k$, so that property $CCP(p, f, \varepsilon)$ is satisfied. We proceed inductively: we apply the Claim to the set $R = V_k$ and define Q_k to be a set from $\mathcal{S}_{V_k,\emptyset}$. If we have defined set Q_i, for $1 < i \leq k$, we apply the Claim to the set $R = V_{i-1}$ and define Q_{i-1} to be a set in $\mathcal{S}_{V_{i-1},Q_i}$ including set

Q_i. The inductive proof shows that the Q_is are well defined and that graph G satisfies property $CCP(p, f, \varepsilon)$. More precisely, the following invariant holds after construction of set Q_i:

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

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

5.2.2.2 Sets of Permutations

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

Let Υ and Ψ, $\Upsilon \subseteq \Psi$, be two sets containing permutations from S_t. For every σ in S_t, let $\sigma \circ \Upsilon$ denote the set of permutation $\{\sigma \circ \pi : \pi \in \Upsilon\}$. For given permutation π, let (d)-$\textsc{lrm}(\pi)$ denote the number of d-left-to-right maxima in π. Now we define the notion of *surfeit*. (We will show that *surfeit* relates to the redundant activity in our algorithms, i.e., “overdone” activity, or literally “surfeit”.) For a given Υ and permutation $\sigma \in S_t$, let $(d, |\Upsilon|)$-Surf(Υ, σ) be equal to $\sum_{\pi\in\Upsilon}(d)$-LRM$(\sigma^{-1} \circ \pi)$. We then define the (d, q)-surfeit of set Ψ as (d, q)-Surf(Ψ) = $\max\{(d, q)\text{-Surf}(\Upsilon, \sigma) : \Upsilon \subseteq \Psi \wedge |\Upsilon| = q \wedge \sigma \in S_t\}$.

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

Lemma 5.9 Let Υ be a set of q random permutations on set $\{1, \ldots, t\}$. For every fixed positive integer d, the probability that (d, q)-Surf$(\Upsilon, \mathbf{e}_t) > t \ln t + 10qd \ln(t + p)$ is at most $e^{-[t \ln t+9qdH_{t+p}] \ln(9/e)}$.

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

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

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

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

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

First notice that for every $\pi \in \Upsilon$ and every $i = 1, \ldots, d$, $\pi(i)$ is d-lrm. Second, observe that $\mathbb{E}\left[\sum_{\pi\in\Upsilon} \sum_{i=d+1}^{t} X(\pi, i)\right] = qd \cdot \sum_{i=d+1}^{t} \frac{1}{i} = qd(H_t - H_d)$. We use Chernoff bound (see [6])

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

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

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

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

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

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

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

This completes the proof of the lemma. □

Theorem 5.10 For a random set of p permutations Ψ from S_t, the event

"for every positive integers d and $q \leq p$, (d,q)-Surf$(\Psi) > t\ln t + 10qd\ln(t+p)$"

holds with probability at most $e^{-t\ln t\cdot\ln(9/e^2)}$.

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

First notice, that if Υ is a random set of permutation, then for arbitrary permutation σ on set $\{1, \ldots, t\}$, set $\sigma^{-1} \circ \Upsilon$ is also a random set of permutation, since composition with a permutation is a bijective operation on sets of q permutations. Consequently, by Lemma 5.9, (d, q)-Surf$(\Upsilon, \sigma) > t \ln t + 10qd \ln(t + p)$ holds with probability at most $e^{-[t \ln t + 9qdH_{t+p}] \ln(9/e)}$.

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

$$\begin{aligned} t! \cdot \binom{p}{q} \cdot e^{-[t \ln t + 9qdH_{t+p}] \ln(9/e)} &\leq e^{t \ln t + q \ln(ep/q) - [t \ln t + 9qdH_{t+p}] \ln(9/e)} \\ &\leq e^{-[t \ln t + 8qdH_{t+p}] \ln(9/e^2)} . \end{aligned}$$

It follows, that the probability of event:

“for every d and q, (d, q)-Surf$(\Psi) > t \ln t + 10qd \ln(t + p)$”,

is at most

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

for $p \geq 1$ and $t \geq 3$. □

Using the probabilistic method [6] we obtain the following result.

Corollary 5.11 There is a set of p permutations Ψ from S_t such that, for every positive integers d and $q \leq p$, (d, q)-Surf$(\Psi) \leq t \ln t + 10qd \ln(t + p)$.

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

5.2.3 The Gossip Algorithm

Our new gossiping algorithm, called GOSSIP$_\varepsilon$, improves on the algorithm of [21]. The improvement is obtained by using the better properties of communication graphs described in Lemma 5.8, the set of permutations with certain properties stated in Corollary 5.11, and by using many epochs instead of the two epochs in [21] (in [21] they refer to epochs as phases). Moreover, the communication graphs we consider have dynamically changing degree, as opposed to [21] that they consider graphs with fixed degree. The challenges motivating our techniques are: (i) how to assure low communication during every epoch, and (ii) how to switch between epochs without a "huge complexity hit".

5.2.3.1 Description of Algorithm GOSSIP$_\varepsilon$

Suppose constant $0 < \varepsilon < 1/3$ is given. The algorithm proceeds in a loop that is repeated until each non-faulty processor v learns either the rumor of every processor w or that w has failed. A single iteration of the loop is called an *epoch*. The algorithm terminates after $\lceil 1/\varepsilon \rceil - 1$ epochs. Each of the first $\lceil 1/\varepsilon \rceil - 2$ epochs consists of $\alpha \log^2 p$ *phases*, where α is such that $\alpha \log^2 p$ is the smallest integer that is larger than $341 \log^2 p$. Each phase is divided into two *stages*, the *update* stage, and the *communication* stage. In the update stage processors update their local knowledge regarding other processors' rumor (known/unknown) and condition (failed/operational) and in the communication stage processors exchange their local knowledge (more momentarily). We say that processor v *heard about processor* w if either v knows the rumor of w or it knows that w has failed. Epoch $\lceil 1/\varepsilon \rceil - 1$ is the terminating epoch where each processor sends a message to all the processors that it haven't heard about, requesting their rumor.

Iterating epochs

for $\ell = 1$ **to** $\lceil 1/\varepsilon \rceil - 2$ do
  if BUSY is empty **then**
    set $status$ to `idle`;
  NEIGHB= $\{v : v \in$ ACTIVE $\wedge\ v \in N_{G_\ell}\}$;
  repeat $\alpha \log^2 p$ times
    update stage;
    communication stage;

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

update stage;
if $status =$ `collector` **then**
  send $\langle$ACTIVE, BUSY, RUMORS, `call`$\rangle$ to
    each processor in WAITING;
receive messages;
send $\langle$ ACTIVE, BUSY, RUMORS, `reply`$\rangle$ to
  each processor in ANSWER;
receive messages;
update RUMORS;

Figure 2: Algorithm GOSSIP$_\varepsilon$. Code for processor v.

The pseudocode of the algorithm is given in Figure 2 (we assume, where needed, that every **if-then** has an implicit **else** clause containing the necessary number of no-ops to match the length of the code in the **then** clause).

Local knowledge and messages. Initially each processor v has its $rumor_v$ and permutation π_v from a set Ψ of permutations on $[p]$, such that Ψ satisfies the thesis of Corollary 5.11. Moreover, each processor v is associated with the variable $status_v$. Initially $status_v =$ `collector` (and we say that v is a collector), meaning that v has not heard from all processors yet. Once v hears from all other processors, then $status_v$ is set to `informer` (and we say that v is an informer), meaning that now v will inform the other processors of its status and knowledge. When processor v learns that all non-faulty processors w also have $status_w =$ `informer` then at the beginning of the next epoch, $status_v$ becomes `idle` (and we say that v idles), meaning that v idles until termination, but it might send responses to messages (see call-messages below).

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

- List ACTIVE$_v$: it contains the pids of the processors that v considers to be non-faulty. Initially, list ACTIVE$_v$ contains all p pids.

- List BUSY_v: it contains the pids of the processors that v consider as collectors. Initially list BUSY_v contains all pids, *permuted according to* π_v.

- List WAITING_v: it contains the pids of the processors that v did not hear from. Initially list WAITING_v contains all pids except from v, *permuted according to* π_v.

- List RUMORS_v: it contains pairs of the form $(w, rumor_w)$ or $(w, \perp)$. The pair $(w, rumor_w)$ denotes the fact that processor v knows processor w's rumor and the pair $(w, \perp)$ means that v does not know w's rumor, but it knows that w has failed. Initially list RUMORS_v contains the pair $(v, rumor_v)$.

A processor can send a message to any other processor, but to lower the message complexity, in some cases (see communication stage) we require processors to communicate according to a conceptual communication graph G_ℓ, $\ell \leq \lceil 1/\varepsilon \rceil - 2$, that satisfies property $CCP(p, p - p^{1-\ell\varepsilon}, \varepsilon)$ (see Definition 5.4 and Lemma 5.8). When processor v sends a message m to another processor w, m contains lists ACTIVE_v, BUSY_v RUMORS_v, and the variable $type$. When $type =$ `call`, processor v requires an answer from processor w and we refer to such message as a *call-message*. When $type =$ `reply`, no answer is required—this message is sent as a response to a call-message.

We now present the sets maintained by processor v.

- Set ANSWER_v: it contains the pids of the processors that v received a call-message. Initially set ANSWER_v is empty.

- Set CALLING_v: it contains the pids of the processors that v will send a call-message. Initially CALLING_v is empty.

- Set NEIGHB_v: it contains the pids of the processors that are in ACTIVE_v and that according to the communication graph G_ℓ, for a given epoch ℓ, are neighbors of v

($\text{NEIGHB}_v = \{w : w \in \text{ACTIVE}_v \wedge w \in N_{G_\ell}(v)\}$). Initially, NEIGHB_v contains all neighbors of v (all nodes in $N_{G_1}(v)$).

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

- First sub-stage: every processor v that is either a collector or an informer (i.e., $status_v \neq$ `idle`) sends message $\langle \text{ACTIVE}_v, \text{BUSY}_v, \text{RUMORS}_v, \texttt{call}\rangle$ to every processor in CALLING_v. The idle processors do not send any messages in this sub-stage.
- Second sub-stage: all processors (collectors, informers and idling) collect the information sent to by the other processors in the previous sub-stage. Specifically, processor v collects lists ACTIVE_w, BUSY_w and RUMORS_w of every processor w that received a call-message from and v inserts w in set ANSWER_v.
- Third sub-stage: every processor (regardless of its status) responds to each processor that received a call-message from. Specifically, processor v sends message $\langle \text{ACTIVE}_v$, BUSY_v, RUMORS_v, $\texttt{reply}\rangle$ to the processors in ANSWER_v and empties ANSWER_v.
- Fourth sub-stage: the processors receive the responses to their call-messages.

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

(r1) Updating BUSY_v or RUMORS_v: For every processor w in CALLING_v (i) if v is an informer, it removes w from BUSY_v, (ii) if v is a collector and RUMORS_w was included in one of the messages that v received, then v adds the pair $(w, rumor_w)$ in RUMORS_v

and, (iii) if v is a collector but RUMORS_w was not included in one of the messages that v received, then v adds the pair $(w, \perp)$ in RUMORS_v.

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

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

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

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

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

Terminating epoch. Epoch $\lceil 1/\varepsilon \rceil - 1$ is the last epoch of the algorithm. In this epoch, each processor v updates its local information based on the messages it received in the last communication stage of epoch $\lceil 1/\varepsilon \rceil - 2$. If after this update processor v is still a collector, then it sends a call-message to every processor that is in WAITING_v (containing pids of the processors whose rumor v does not know or processors that failed). Then every processor receives the call-messages sent by the other processors. Next, every processor that received a call-message sends its local knowledge to the sender. Finally each processor v updates RUMORS_v based on any received information.

5.2.3.2 Correctness of Algorithm $\text{GOSSIP}_\varepsilon$

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

Lemma 5.12 In any execution of algorithm $\text{GOSSIP}_\varepsilon$, if processors v and w are non-faulty by the end of any epoch $\ell < \lceil 1/\varepsilon \rceil - 1$, then w is in ACTIVE_v and vice-versa.

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

Initially all processors (including w) are in ACTIVE_v. Consider phase s of epoch 1 (for simplicity assume that s is not the last phase of epoch 1). By the update rule, a processor w is removed from ACTIVE_v if v is not idle and (a) during the communication stage of phase s, w is not in NEIGHB_v and v received a message from a processor v' so that w is not in $\text{ACTIVE}_{v'}$, (b) during the communication stage of phase s, w is in NEIGHB$_v$ and v did not receive a message from w, or (c) v sent a call-message to w in the communication stage of phase s of epoch 1 and v did not receive a response from w in the same stage.

Case (c) is not possible: Since w is non-faulty in all phases s of epoch 1, w receives the call-message from v in the communication stage of phase s and adds v in ANSWER$_w$. Then, processor w sends a response to v in the same stage. Hence v does not remove w from ACTIVE$_v$. Case (b) is also not possible: Since w is non-faulty and w is in NEIGHB$_v$, by the properties of the communication graph G_1, v is in NEIGHB$_w$ as well (and since v is non-faulty). From the description of the first sub-stage of the communication stage, if $status_w \neq$ `idle`, w sends a message to its neighbors, including v. If $status_w =$ `idle`, then w will not send a message to v in the first sub-stage, but it will send a reply to v' call-message in the third sub-stage. Therefore, by the end of the communication stage, v has received a message from w and hence it does not remove w from ACTIVE$_v$. Neither case (a) is possible: This follows inductively, using points (b) and (c): no processor will remove w from its set of active processors in a phase prior to s and hence v does not receive a message from any processor v' so that w is not in ACTIVE$_{v'}$.

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

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

Lemma 5.13 In any execution of algorithm $\text{GOSSIP}_\varepsilon$ and any epoch $\ell < \lceil 1/\varepsilon \rceil - 1$, if processors v and w are non-faulty by the end of epoch ℓ and $status_w = \texttt{collector}$, then w is in BUSY_v.

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

Initially all processors w have status `collector` and w is in BUSY_v ($\text{CALLING}_v \setminus \text{NEIGHB}_v$ is empty). Consider phase s of epoch 1. By the update rule, a processor w is removed from BUSY_v if (a) at the beginning of the update stage of phase s, v is an informer and w is in CALLING_v, or (b) during the communication stage of phase s, v receives a message from a processor v' so that w is not in $\text{BUSY}_{v'}$.

Case (a) is not possible: Since v is an informer and w is in CALLING_v at the beginning of the update stage of phase s, this means that in the communication stage of phase $s-1$, processor v was already an informer and it sent a call-message to w. In this case, w would receive this message and it would become an informer during the update stage of phase s. This violates the assumption of the lemma. Case (b) is also not possible: For w not being in $\text{BUSY}_{v'}$ it means that either (i) in some phase $s' < s$, processor v' became an informer and sent a call-message to w, or (ii) during the communication stage of a phase $s'' < s$, v' received a message from a processor v'' so that w was not in $\text{BUSY}_{v''}$. Case (i) implies that in phase $s' + 1$, processor w becomes an informer which violates the assumption of the lemma. Using inductively case (i) it follows that case (ii) is not possible either.

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

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

Lemma 5.14 In any execution of algorithm $\text{GOSSIP}_\varepsilon$ and any epoch $\ell < \lceil 1/\varepsilon \rceil - 1$,

(i) if processors v and w are non-faulty by the end of epoch ℓ and w is not in WAITING_v, then $(w, rumor_w)$ is in RUMORS_v, and (ii) if processor v is non-faulty by the end of epoch ℓ and $(w, \perp)$ is in RUMORS_v, then w is not in ACTIVE_v.

Proof: We first prove part (i) of the lemma. Consider processors v and w that are non-faulty by the end of epoch ℓ and that w is not in WAITING_v. The proof proceeds by induction on the number of epochs.

Initially w is in WAITING_v and RUMORS_v contains only the pair $(v, rumor_v)$. Consider phase s of epoch 1. By the update rule, processor w is removed from WAITING_v if during the update stage of phase s, either $(w, rumor_w)$ or $(w, \perp)$ is in RUMORS_v. In order for $(w, rumor_w)$ to be in RUMORS_v by phase s one of the following must be true: (a) Processor v sent a call-message to processor w in the communication stage of phase $s - 1$ and v received a response from w, (b) During the communication stage of phase $s - 1$ processor v received a message from processor v' so that $(w, rumor_w)$ is in $\text{RUMORS}_{v'}$.

Case (a) is possible, since w is non-faulty. In case (b), in order for processor v' to know the rumor of w it must either learned it (b$'$) from w or (b$''$) from some other processor v'' in a phase

$s' < s - 1$. Case (b$'$) shows trivially that case (b) is possible. For case (b$''$) to be possible, it must be the case that either v'' learned the rumor of w from w or some other node v''' in a phase $s'' < s'$. Using case (b$'$) inductively, it follows that case (b$''$) is possible, and thus, case (b) is possible. Hence, if by the end of epoch 1, w is not in WAITING_v, then $(w, rumor_w)$ is in RUMORS_v.

Now assuming that part (i) of the lemma holds by the end of epoch $\ell - 1$, we would like to show that it also holds by the end of epoch ℓ. This follows from the inductive hypothesis and the fact that no processor identifier is ever added in WAITING_v and no pair of the form $(w, rumor_w)$ is removed from RUMORS_v.

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

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

Theorem 5.15 By the end of epoch $\lceil 1/\varepsilon \rceil - 1$ of any execution of algorithm $\text{GOSSIP}_\varepsilon$, every non-faulty processor v either knows the rumor of processor w or it knows that w has failed.

Proof: Consider a processor v that is non-faulty by the end of epoch $\lceil 1/\varepsilon \rceil - 1$. In the update stage of epoch $\lceil 1/\varepsilon \rceil - 1$ processor v updates it local knowledge based on the knowledge it had in the previous epochs and the new information it obtained in the communication stage of the last phase of epoch $\lceil 1/\varepsilon \rceil - 2$. Lemmas 5.12, 5.13, and 5.14 guarantee that this knowledge does not contain false information.

If after this last update, processor v is still a collector, meaning that v did not hear from all processors yet, according to the description of the algorithm, processor v will send a call-message to the processors whose pid is still in WAITING_v (by Lemma 5.14 and the update rule, it follows that list WAITING_v contains all processors that v did not hear from yet). Then all non-faulty processors w receive the call-message of v and then they respond to v. Then v receives these responses. Finally v updates list RUMORS_v accordingly: if a processor w responded to v's call-message (meaning that v now learns the rumor of w), then v adds $(w, rumor_w)$ in RUMORS_v. If w did not respond to v's call-message, and $(w, rumor_w)$ is not in RUMORS_v (it is possible for processor v to learn the rumor of w from some other processor v' that learned the rumor of w before processor w failed), then v knows that w has failed and adds $(w, \perp)$ in RUMORS_v.

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

Note from the above that the correctness of algorithm $\text{GOSSIP}_\varepsilon$ does not depend on whether the set of permutations Ψ satisfy the conditions of Corollary 5.11. The algorithm is correct for any set of permutations of $[p]$.

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

Consider some set V_ℓ, $|V_\ell| \geq p^{1-\ell\varepsilon}$, of processors that are not idle at the beginning of epoch ℓ and do not fail by the end of epoch ℓ. Let $Q_\ell \subseteq V_\ell$ be such that $|Q_\ell| \geq |V_\ell|/7$ and the diameter of the subgraph induced by Q_ℓ is at most $31 \log p$. Q_ℓ exists because of Lemma 5.8 applied to graph G_ℓ and set V_ℓ (chains have size 2).

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

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

Lemma 5.16 The total number of progress-messages sent by processors in Q_ℓ from the beginning of epoch ℓ until the first processor in Q_ℓ will have its list WAITING (or list BUSY) empty, is at most $(d, |Q_\ell|)$-Surf(Ψ).

Proof: Fix Q_ℓ and consider some permutation $\sigma \in S_p$ that satisfies the following property: "Consider $i < j \leq p$. Let τ_i (τ_j) be the time step in epoch ℓ where some processor in Q_ℓ hears about $\sigma(i)$ ($\sigma(j)$) the first time among the processors in Q_ℓ. Then $\tau_i \leq \tau_j$." (We note that it is not difficult to see that for a given Q_ℓ we can always find $\sigma \in S_p$ that satisfies the above property.) We consider only the subset $\Upsilon \subseteq \Psi$ containing permutations of indexes from set Q_ℓ. To show the lemma we prove that the number of messages sent by processors from Q_ℓ is at most $(d, |\Upsilon|)$-Surf$(\Upsilon, \sigma) \leq (d, |Q_\ell|)$-Surf$(\Psi)$. Suppose that processor $v \in Q_\ell$ sends a progress-message to processor w. It follows from the diameter of Q_ℓ and the size of set CALL in epoch ℓ, that none of processor $v' \in Q_\ell$ had sent a progress-message to w

before $31 \log p$ phases, and consequently position of processor w in permutation π_v is at most $d - |\text{CALL}| \leq d - p^{(\ell+1)\varepsilon}$ greater than position of w in permutation $\pi_{v'}$.

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

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

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

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

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

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

Using Lemma 5.16 and Corollary 5.11 we show the following:

Lemma 5.17 In any execution of algorithm GOSSIP$_\varepsilon$, the invariant I_ℓ holds for any epoch $\ell = 1, \ldots, \lceil 1/\varepsilon \rceil - 2$.

Proof: For $p = 1$ it is obvious. Assume $p > 1$. We will use Lemma 5.8 and Corollary 5.11. Consider any epoch $\ell < \lceil 1/\varepsilon \rceil - 1$. Suppose to the contrary, that there is a subset V_ℓ of non-faulty processors after the end of epoch ℓ such that each of them has status either `collector` or `informer` and $|V_\ell| > p^{1-\ell\varepsilon}$. Since G_ℓ satisfies $CCP(p, p - p^{1-\ell\varepsilon}, \varepsilon)$, there is a set $Q_\ell \subseteq V_\ell$ such that $|Q_\ell| \geq |V_\ell|/7 > p^{1-\ell\varepsilon}/7$ and the diameter of the subgraph induced by Q_ℓ is at most $31 \log p$. Applying Lemma 5.16 and Corollary 5.11 to the set Q_ℓ, epoch ℓ, $t = p$, $q = |Q_\ell|$ and $d = 31 p^{(\ell+1)\varepsilon} \log p$, we obtain that the total number of messages sent until some processor $v \in Q_\ell$ has list BUSY$_v$ empty, is at most

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

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

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

Notice that since no processor in Q_ℓ has status `idle` in epoch ℓ, each of them sends in every phase of epoch ℓ at most $|\text{CALL}| \leq p^{(\ell+1)\varepsilon}$ progress-messages. Consequently the total number of phases in epoch ℓ until some of the processors in Q_ℓ has its list BUSY empty, is at most $\dfrac{341|Q_\ell|p^{(\ell+1)\varepsilon}\log^2 p}{|Q_\ell|p^{(\ell+1)\varepsilon}} \leq 341\log^2 p$.

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

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

Theorem 5.18 Algorithm GOSSIP$_\varepsilon$ solves the *Gossip*$_{\mathcal{A}_S}(p, f)$ problem with time complexity $T = O(\log^2 p)$ and message complexity $M = O(p^{1+3\varepsilon})$.

Proof: First we show the bound on time. Observe that each update and communication stage takes $O(1)$ time. Therefore each of the first $\lceil 1/\varepsilon\rceil - 2$ epochs takes $O(\log^2 p)$ time. The last epoch takes $O(1)$ time. From this and the fact that ε is a constant, we have that the time complexity of the algorithm is in the worse case $O(\log^2 p)$. We now show the bound on messages. From Lemma 5.17 we have that for every $1 \leq \ell < \lceil 1/\varepsilon\rceil - 2$, during epoch $\ell+1$ there are at most $p^{1-\ell\varepsilon}$ processors sending at most $2p^{(\ell+2)\varepsilon}$ messages in every communication stage. The remaining processors are either faulty (hence they do not send any messages) or have status `idle` — these processors only respond to call-messages and their total impact on the message complexity in epoch $\ell+1$ is at most as large as the others. Consequently the message complexity during epoch $\ell+1$ is at most $4(\alpha\log^2 p)\cdot(p^{1-\ell\varepsilon}p^{(\ell+2)\varepsilon}) \leq 4\alpha p^{1+2\varepsilon}\log^2 p \leq$

$4\alpha p^{1+3\varepsilon}$. After epoch $\lceil 1/\varepsilon \rceil - 2$ there are, per $\mathrm{I}_{\lceil 1/\varepsilon \rceil - 2}$, at most $p^{2\varepsilon}$ processors having list WAITING not empty. In epoch $\lceil 1/\varepsilon \rceil - 1$ each of these processors sends a message to at most p processors twice, hence the message complexity in this epoch is bounded by $2p \cdot p^{2\varepsilon}$. From the above and the fact that ε is a constant, we have that the message complexity of the algorithm is $O(p^{1+3\varepsilon})$. □

5.2.4 The Do-All Algorithm based on Gossip

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

5.2.4.1 Description of Algorithm DOALL_ε

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

The pseudocode for a phase of epoch ℓ of the algorithm is given in Figure 3 (again we assume that every **if-then** has an implicit **else** containing no-ops as needed).

Work stage

```
repeat T_ℓ times
  if TASK not empty then
    perform task whose id is first in TASK;
    remove task's id from TASK;
  elseif TASK empty and done = false
    then set done to true;
if TASK empty and done = false then
  set done to true;
```

Gossip stage

```
run GOSSIP_{ε/3} with rumor = (TEMP, PROC, done);
if done = true and done_w = true for all w
received rumor from then
  TERMINATE;
else
  update TASK and PROC;
```

Figure 3: A phase of epoch ℓ of algorithm $\mathrm{DOALL}_{\varepsilon}$. Code for processor v.

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

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

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

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

Gossip stage. Here processors execute algorithm $\text{GOSSIP}_{\varepsilon/3}$ using their local knowledge as the rumor, i.e., for processor v, $rumor_v = (\text{TASK}_v, \text{PROC}_v, done_v)$. At the end of the stage, each processor v updates its local knowledge based on the rumors it received. The **update rule** is as follows: (a) If v does not receive the rumor of processor w, then v learns that w has failed (guaranteed by the correctness of $\text{GOSSIP}_{\varepsilon/3}$). In this case v removes w from PROC_v. (b) If v receives the rumor of processor w, then it compare TASK_v and PROC_v with TASK_w and PROC_w respectively and updates its lists accordingly—it removes the tasks that w knows are already completed and the processors that w knows that have crashed. Note that if TASK_v becomes empty after this update, variable $done_v$ remains `false`. It will be set to `true` in the next work stage. This is needed for the correctness of the algorithm (see Lemma 5.22).

Termination decision. We would like all non-faulty processors to learn that the tasks are done. Hence, it would not be sufficient for a processor to terminate once the value of its *done* variable is set to `true`. It has to be assured that all other non-faulty processors' *done* variables are set to `true` as well, and then terminate. This is achieved as follows: If processor v starts the gossip stage of a phase of epoch ℓ with $done_v =$ `true`, and all rumors it receives suggest that all other non-faulty processors know that all tasks are done (their *done* variables are set to `true`), then processor v terminates. If at least one processor's *done* variable is set to `false`, then v continues to the next phase of epoch ℓ (or to the first phase of epoch $\ell+1$ if the previous phase was the last of epoch ℓ).

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

5.2.4.2 Correctness of Algorithm DOALL$_\varepsilon$

We show that the algorithm DOALL$_\varepsilon$ solves the *Do-All*$_{\mathcal{A}_S}(n, p, f)$ problem correctly, meaning that the algorithm terminates with all tasks performed and all non-faulty processors are aware of this. Note that this is a stronger correctness condition than the one required by the definition of *Do-All*.

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

Lemma 5.19 In any execution of algorithm DOALL$_\varepsilon$, if processors v and w are non-faulty by the end of the gossip stage of phase s of epoch ℓ, then processor w is in PROC$_v$ and vice-versa.

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

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

Lemma 5.20 In any execution of algorithm DOALL_ε, if a task z is not in TASK_v of any processor v at the beginning of the first phase of epoch ℓ, then z has been performed in a phase of one of the epochs $1, 2, \ldots, \ell - 1$.

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

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

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

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

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

Lemma 5.21 In any execution of algorithm DOALL_ε, if processor v enters a work stage of a phase s of epoch ℓ with $done_w = \texttt{false}$, then *sizeof*$({\text{TASK}_v}^{(\ell,s+1)}) <$ *sizeof*$({\text{TASK}_v}^{(\ell,s)})$.

Proof: Let v be a processor that starts the work stage of phase s of epoch ℓ with $done_w =$ $\texttt{false}$. According to the description of the algorithm, the value of variable $done_v$ is initially $\texttt{false}$ and it is set to $\texttt{true}$ only when TASK_v becomes empty. Hence, at the beginning of the work stage of phase s of epoch ℓ there is at least one task identifier in ${\text{TASK}_v}^{(\ell,s)}$, and therefore v performs at least one task. From this and the fact that no tid is ever added in a processor's task list, we get that *sizeof*$({\text{TASK}_v}^{(\ell,s+1)}) <$ *sizeof*$({\text{TASK}_v}^{(\ell,s)})$. $\square$

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

Lemma 5.22 In any execution of algorithm DOALL_ε, for any phase s of epoch ℓ and any processor v, if $done_v$ is set to `true` during phase s and v is non-faulty by the end of phase s, then the algorithm terminates by phase $s + 1$ of epoch ℓ.

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

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

Finally we show the correctness of algorithm DOALL_ε.

Theorem 5.23 In any execution of algorithm DOALL_ε, the algorithm terminates with all tasks performed and all non-faulty processors being aware of this.

Proof: By Lemma 5.19, no non-faulty processor leaves the computation, and by our model at least one processor does not crash ($f < p$). Also from Lemma 5.20 we have that no undone task is removed from the computation. From the code of the algorithm we get that a processor continues performing tasks until its TASK list becomes empty and by Lemma 5.21 we have that local progress is guaranteed. The above in conjunction with the correctness of algorithm GOSSIP$_{\varepsilon/3}$ lead to the conclusion that there exist a phase s of an epoch ℓ and a processor v so that during phase s processor v sets $done_v$ to `true`, all tasks are indeed performed and v survives phase s. By Lemma 5.22 the algorithm terminates by phase $s+1$ of epoch ℓ (or by the first phase of epoch $\ell+1$ if s is the last phase of epoch ℓ). Now, from the definition of T_ℓ it follows that the algorithm terminates after at most $O(\log p)$ epochs: consider epoch $\log p$; $T_{\log p} = \lceil (n + p\log^3 p)/\log p \rceil = \lceil n/\log p + p\log^2 p \rceil$. Recall that each epoch consists of $\beta \log p + 1$ phases. Say that $\beta = 1$. Then, when a processor reaches epoch $\log p$, it can perform all n tasks in this epoch. Hence, all tasks that are not done until epoch $\log p - 1$ are guaranteed to be performed by the end of epoch $\log p$ and all non-faulty processors will know that all tasks have been performed. $\square$

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

5.2.4.3 Analysis of Algorithm DOALL$_\varepsilon$

We now derive the work and message complexities for algorithm DOALL$_\varepsilon$. Our analysis is based on the following terminology. Consider a phase i in epoch ℓ of an execution $\xi \in \mathcal{E}(\text{DOALL}_\varepsilon, \mathcal{A}_S)$. Let $V_i(\xi)$ denote the set of processors that are non-faulty at the beginning of phase i. Let $p_i(\xi) = |V_i(\xi)|$. Let $U_i(\xi)$ denote the set of tasks z such that z is in some list TASK_v, for some $v \in V_i(\xi)$, at the beginning of phase i. Let $u_i(\xi) = |U_i(\xi)|$.

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

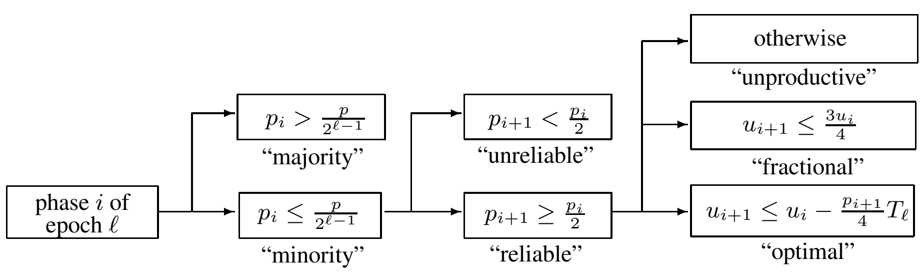

Figure 4: Classification of a phase i of epoch ℓ; execution ξ is implied.

Our goal is to choose a set Ψ of permutations such that for any execution there will be no unproductive and no majority phases. To do this we analyze sets of random permutations, prove certain properties of our algorithm for such sets (in Lemmas 5.24 and 5.25), and finally use the probabilistic method to obtain an existential deterministic solution.

Lemma 5.24 Let Q be a fixed nonempty subset of processors. Then the probability of event "for every execution ξ of algorithm $\text{DOALL}_{\varepsilon}$ such that $V_{i+1}(\xi) \supseteq Q$ and $u_i(\xi) > 0$, the following inequality holds $u_i(\xi) - u_{i+1}(\xi) \geq \min\{u_i(\xi), |Q|T_\ell\}/4$," is at least $1 - 1/e^{\Omega(|Q|T_\ell)}$.

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

Consider the following cases:

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

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

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

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

$$s_i(\xi) - \big(u_i(\xi) - u_i^{(k)}(\xi)\big) > s_i(\xi) - c \geq u_i(\xi)/2 \geq \textit{sizeof}(\text{TASK}_v)/2,$$

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

We need to estimate the probability $\Pr[\sum X_k \ge c]$, where the summation is over all $|Q|T_\ell$ steps of all the processors in Q in the considered process. Consider a sequence $\langle Y_k \rangle$ of independent Bernoulli trials, with $\Pr[Y_k = 1] = 1/2$. Then the sequence $\langle X_k \rangle$ statistically dominates the sequence $\langle Y_k \rangle$, in the sense that $\Pr\big[\sum X_k \ge d\big] \ge \Pr\big[\sum Y_k \ge d\big]$, for any $d > 0$. Note that $\mathbb{E}[\sum Y_k] = |Q|T_\ell/2$ and $c \le \mathbb{E}[\sum Y_k]/2$, hence we can apply Chernoff bound to obtain

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

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

Lemma 5.25 Assume $n \le p^2$. There exists a constant integer $\beta > 0$ such that for every phase i of any epoch ℓ of any execution ξ of algorithm DOALL_ε, if there is a task unperformed by the beginning of phase i then: (a) the probability that phase i is a majority phase is at most $e^{-\Omega(p \log p)}$, and (b) the probability that phase i is a minority reliable unproductive phase is at most $e^{-\Omega(T_\ell)}$.

Proof: The proof is by induction on phase i. For phase 1 claim (a) holds even with the probability 0, since $p \le \frac{p}{2^{\ell-1}}$. We prove claim (b). Consider executions such that phase 1 is minority reliable. We can partition these executions according to the following equivalence relation: executions ξ_1 and ξ_2 are in the same class iff $V_2(\xi_1) = V_2(\xi_2)$. Consider a set of processors Q of size at least $p/2$, and any execution ξ such that $V_2(\xi) = Q$. By Lemma 5.24 applied to Q and phase 1 we get that the probability that $u_1(\xi) - u_2(\xi) < \min\{u_1(\xi), |Q|T_1\}/4$ is at most

$e^{-\Omega(|Q|T_1)} \leq e^{-\Omega(p \log p)}$. There are at most 2^p different groups of executions represented by different sets Q, hence the probability that for every execution ξ, phase 1 is a minority reliable unproductive phase is at most $2^p \cdot e^{-\Omega(p \log p)} = e^{-\Omega(p \log p)} \leq e^{-\Omega(T_1)}$. Thus claim (b) holds for phase 1. Note that so far we have not obtained any bounds on β.

Suppose that claims (a) and (b) hold for every phase up to $i-1$, where $i-1 \geq 1$ and there is an unperformed task at the beginning of phase $i-1$. We prove that if there is an unperformed task at the beginning of phase i then claims (a) and (b) hold for phase i.

Assume that phase i belongs to epoch ℓ, for some $\ell \geq 1$. First we group executions ξ such that phase i is a majority phase in ξ, according to the following equivalence relation: execution ξ_1 and ξ_2 are in the same class iff $V_{i+1}(\xi_1) = V_{i+1}(\xi_2)$. Every such equivalence class is represented by some set of processors Q of size greater than $\frac{p}{2^{\ell-1}}$, such that for every execution ξ in this class we have $V_{i+1}(\xi) = Q$. We now define conditions for β that keep claim (a) satisfied.

Claim: There is a constant $\beta > 0$ such that for any execution ξ in the class represented by Q, where $|Q| > \frac{p}{2^{\ell-1}}$, all tasks were performed by the end of epoch $\ell - 1$ with probability $e^{-\Omega(p \log p)}$.

We now prove the Claim. Consider an execution ξ from a class represented by Q. Consider all steps taken by processors in Q during phase j of epoch $\ell - 1$. By Lemma 5.24, since $V_{j+1}(\xi) \supseteq Q$, we have that the probability of event "if $u_j(\xi) > 0$ then $u_j(\xi) - u_{j+1}(\xi) \geq \min\{u_j(\xi), |Q|T_{\ell-1}\}/4$," is at least $1 - 1/e^{\Omega(|Q|T_{\ell-1})}$. If the above condition is satisfied we call phase j productive (by similarity to names optimal and fractional, but the difference is that these names are used only for minority phases, but now we use it according to the progress made by processors in Q), and this happens with probability $1 - 1/e^{\Omega(|Q|T_{\ell-1})}$. Since the total number of tasks is n, we have that the number of productive phases during epoch $\ell-1$ sufficient to perform all tasks using only processors in Q is either at most $\frac{n}{|Q|T_{\ell-1}/4} \leq \frac{n}{n/(4\log p)} = 4\log p$, or, since $n \leq p^2$, is at most $\log_{1/4} n = O(\log p)$.

Therefore there are a total of $O(\log p)$ productive phases, which is sufficient to perform all tasks. Furthermore, every phase in epoch $\ell-1$ is productive. Hence, all tasks are performed by processors in Q during $\beta \log p$ phases, for some constant $\beta > 0$, of epoch $\ell-1$ with probability $1 - O(\log p) \cdot e^{-\Omega(|Q|T_{\ell-1})} = 1 - e^{-\Omega(p\log p)}$. Consequently all processors terminate by the end of phase $\beta \log p + 1$ with probability $1 - e^{-\Omega(p \log p)}$. This follows by the correctness of the gossip algorithm and the argument of Lemma 5.22, since epoch $\ell - 1$ lasts $\beta \log p + 1$ phases and processors in Q are non-faulty at the beginning of epoch ℓ. This completes the proof of the Claim.

There are at most 2^p of possible sets Q of processors, hence by the Claim the probability that phase i is a majority phase is at most $2^p \cdot e^{-\Omega(p\log p)} \leq e^{-\Omega(p \log p)}$, which proves claim (a) for phase i.

Now we prove claim (b) for phase i. Consider executions such that phase i in epoch ℓ is a minority reliable phase. Similarly as above, we partitions executions according to the following equivalence relation: executions ξ_1 and ξ_2 are in the same class if there is set Q such that $H = V_{i+1}(\xi_1) = V_{i+1}(\xi_2)$. Set Q is a representative of a class. By Lemma 5.24 applied to phase i and set Q we obtain that the probability that phase i is unproductive for every execution ξ such that $V_{i+1}(\xi) = Q$ is $e^{-\Omega(|Q|T_\ell)}$. Hence the probability that for any execution ξ phase i is a minority reliable unproductive phase is at most

$$\sum_{x=1}^{p/2^{\ell-1}} \binom{p}{x} \cdot e^{-\Omega(xT_\ell)} \leq \sum_{x=1}^{p/2^{\ell-1}} 2^{x\log p} \cdot e^{-\Omega(xT_\ell)} \leq e^{-\Omega(T_\ell)} \ ,$$

and claim (b) is shown for phase i. □

Recall that epoch ℓ consists of $\beta \log p + 1$ phases for some $\beta > 0$ and that $T_\ell = \lceil \frac{n+p\log^3 p}{(p/2^\ell)\log p} \rceil$. Also by the correctness proof of algorithm DOALL$_\varepsilon$ (Theorem 5.23), the algorithm terminates in at most $O(\log p)$ epochs, hence, the algorithm terminates in at most $O(\log^2 p)$ phases. Let g_ℓ be the number of steps that each gossip stage takes in epoch ℓ, i.e., $g_\ell = \Theta(\log^2 p)$.

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

Theorem 5.26 There is a set of permutations Ψ and a constant integer $\beta > 0$ such that algorithm DOALL$_\varepsilon$, using permutations from Ψ, solves the *Do-All*$_{\mathcal{A}_S}(n, p, f)$ problem with work $S = O(n + p\log^3 p)$ and message complexity $M = O(p^{1+2\varepsilon})$.

Proof: We show that for any execution $\xi \in \mathcal{E}(\text{DOALL}_\varepsilon, \mathcal{A}_S)$ that solves the *Do-All*$_{\mathcal{A}_S}(n, p, f)$ problem there exists a set of permutations Ψ and an integer $\beta > 0$ so that the complexity bounds are as desired. We consider two cases:

Case 1: $n \leq p^2$. Consider phase i of epoch ℓ of execution ξ for randomly chosen set of permutations Ψ. We reason about the probability of phase i belonging to one of the classes illustrated in Figure 4, and about the work that phase i contributes to the total work incurred in the execution, depending on its classification. From Lemma 5.25(a) we get that phase i may be a majority phase with probability $e^{-\Omega(p\log p)}$ which is a very small probability. More precisely, the probability that for a set of permutations Ψ, in execution ξ obtained for Ψ some phase i is a majority phase, is $O(\log^2 p \cdot e^{-\Omega(p\log p)}) = e^{-\Omega(p\log p)}$, and consequently using the probabilistic method argument we obtain that for almost any set of permutations Ψ there is no execution in which there is a majority phase.

Therefore, we focus on minority phases that occur with high probability (per Lemma 5.25(a)). We can not say anything about the probability of a minority phase to be a reliable or unreliable, since this depends on the specific execution. Note however, that by definition, we cannot have more than $O(\log p)$ unreliable minority phases in any execution ξ (at least one processor must remain operational). Moreover, the work incurred in an unreliable minority phase i of an epoch ℓ in any execution ξ is bounded by $O(p_i(\xi) \cdot (T_\ell + g_\ell)) = O(\frac{p}{2^{\ell-1}} \cdot (\frac{n+p\log^3 p}{\frac{p}{2^\ell}\log p} + \log^2 p)) = O(\frac{n}{\log p} + p\log^2 p)$. Thus, the total work incurred by all unreliable minority phases in any execution ξ is $O(n + p\log^3 p)$.

From Lemmas 5.24 and 5.25(b) we get that a reliable minority phase may be fractional or optimal with high probability $1 - e^{-\Omega(T_\ell)}$, whereas it may be unproductive with very small probability $e^{-\Omega(T_\ell)} \leq e^{-\log^2 p}$. Using a similar argument as for majority phases, we get that for almost all sets of permutations Ψ (probability $1 - O(\log^2 p \cdot e^{-\Omega(T_\ell)}) \geq 1 - e^{-\Omega(T_\ell)}$) and for every execution ξ, there is no minority reliable unproductive phase. The work incurred by a fractional phase i of an epoch ℓ in any execution ξ is bounded by $O(p_i(\xi) \cdot (T_\ell + g_\ell)) = O(\frac{n}{\log p} + p\log^2 p)$. Also note that by definition, there can be at most $O(\log_{3/4} n)$ ($= O(\log p)$ since $n \leq p^2$) fractional phases in any execution ξ and hence, the total work incurred by all fractional reliable minority phases in any execution ξ is $O(n + p\log^3 p)$. We now consider the optimal reliable minority phases for any execution ξ. Here we have an optimal allocation of tasks to processors in $V_i(\xi)$. By definition of optimality, in average one task in $U_i(\xi) \setminus U_{i+1}(\xi)$ is performed by at most *four* processors from $V_{i+1}(\xi)$, and by definition of reliability, by at most *eight* processors in $V_i(\xi)$. Therefore, in optimal phases, each unit of work spent on performing a task results to a unique task completion (within a constant overhead), for any execution ξ. It therefore follows that the work incurred in all optimal reliable minority phases is bounded by $O(n)$ in any execution ξ.

Therefore, from the above we conclude that when $n \leq p^2$, for random set of permutations Ψ the work complexity of algorithm DOALL_ε executed on such set Ψ is $S = O(n + p\log^3 p)$ with probability $1 - e^{-\Omega(p \log p)} - e^{-\Omega(T_\ell)} = 1 - e^{-\Omega(T_\ell)}$ (the probability appears only from analysis of majority and unproductive reliable minority phases). Consequently such set Ψ exists. Also, from Lemma 5.25 and the above discussion, $\beta > 0$ exists. Finally, the bound on messages using selected set Ψ and constant β is obtained as follows: there are $O(\log^2 p)$ executions of gossip stages. Each gossip stage requires $O(p^{1+\varepsilon})$ messages (message complexity of one instance of $\text{GOSSIP}_{\varepsilon/3}$). Thus, $M = O(p^{1+\varepsilon}\log^2 p) = O(p^{1+2\varepsilon})$.

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

5.2.4.4 Sensitivity Training and Failure-Sensitive Analysis

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

Algorithm $\text{GOSSIP}'_\varepsilon$. Algorithm $\text{GOSSIP}'_\varepsilon$ is a modified version of algorithm $\text{GOSSIP}_\varepsilon$. In particular, algorithm $\text{GOSSIP}'_\varepsilon$ contains a new epoch, called epoch 0. Epochs $1, \ldots, \lceil 1/\varepsilon \rceil - 1$

are the same epochs as in algorithm $\textsc{Gossip}_\varepsilon$. Assume for simplicity of presentation that $p/\log^2 p$ is an even integer. Epoch 0 is similar to the epoch 1 of algorithm $\textsc{Gossip}_\varepsilon$, except from the following:

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

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

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

Theorem 5.27 There exist constant α' and set Ψ such that algorithm GOSSIP$'_{\varepsilon}$, using set Ψ, solves the *Gossip*$_{\mathcal{A}_S}(p, f)$ problem with time complexity $T = O(\log^2 p)$ and message complexity $M = O(p)$ when $f \leq \frac{p}{\log^2 p}$, and with $T = O(\log^2 p)$ and $M = O(p^{1+3\varepsilon})$ otherwise.

Proof: First we consider the case where there are at most $\frac{p}{\log^2 p}$ failures by the end of epoch 0. Let $Q' \subseteq V'$ be a set of processors such that $|Q'| \geq |V'|/2 \geq \frac{p}{\log^2 p}$. By PROPERTY $\mathcal{R}(|V'|, |V'|/2)$ there exists $Q \subseteq Q'$ such that $|Q| \geq |Q'|/7$ and the diameter of graph G_Q is at most $31 \log p$. Consider all executions $\xi \in \mathcal{E}(\text{GOSSIP}'_{\varepsilon}, \mathcal{A}_S)$ such that every processor in Q' is not failed by the end of epoch 0, and choose Ψ randomly. We may look at the process of collecting rumors by processors in Q (when every processor in Q works as a `collector`) as performing tasks: if a rumor of processor w (or information that processor w is failed), for every processor w, is known by some processor in Q then we say that task

w is performed. We partition the execution into consecutive *blocks*, each containing $31 \log p$ consecutive phases. Notice that during each block all processors in Q exchange information between themselves, by definition of Q. We may use Lemma 5.24 to bound progress: the probability that "for every considered execution ξ (such that all processors in Q are not failed at the end of epoch 0) after every consecutive block in epoch 0 the number of rumors unknown by processors in Q decreases either by $(3/4)|Q| \log p$ or by factor $3/4$ " is $1 - e^{-\Omega(|Q| \log p)}$. Consequently, for every considered execution ξ, $O(\frac{p}{|Q| \log p} + \log_{3/4} p) = O(\log p)$ number of blocks are sufficient to collect all rumors by processors in Q, with probability at least $1 - \log p \cdot e^{-\Omega(|Q| \log p)} \geq 1 - e^{-\Omega(|Q| \log p)}$. Using the probabilistic method we choose one such Ψ, which additionally satisfies the thesis of Theorem 5.10 (to assure that Ψ is good also for the other cases in this proof) and constant α' follows from the fact that $O(\log p)$ blocks, each of $31 \log p$ phases, suffices to collect all rumors by processors in Q for every execution ξ.

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

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

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

Using the same argument as in the proof of Theorem 5.18 and by the fact that Ψ was chosen to satisfy the thesis of Theorem 5.10, we obtain that the message complexity during execution of $\text{GOSSIP}'_\varepsilon$ is $O(p^{1+2\varepsilon}\log^3 p) = O(p^{1+3\varepsilon})$, which together with $O(p)$ messages sent in epoch 0 yields the thesis of the theorem, with respect to message complexity. The time complexity yields from the fact that epoch 0 runs $O(\log^2 p)$ phases, and the remaining epochs run also for $O(\log^2 p)$ phases. $\square$

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

Epoch -1 of algorithm $\text{DOALL}'_\varepsilon$ uses the check-pointing algorithm from [28], where the check-pointing and the synchronization procedures are taken from [44]. We refer to the algorithm used in epoch -1 as algorithm DGMY. The goal of using this algorithm in epoch -1 is to solve *Do-All* with work $O(n+p(f+1))$ and communication $O(fp^\varepsilon+p\min\{f+1,\log p\})$ if the number of failures is small, mainly concerning the case $f \leq \log^3 p$. Hence, in epoch -1, we execute DGMY *only until* step $a \cdot (n/p + \log^3 p)$, for some constant a such that the early-stopping condition of DGMY holds for every $f \leq \log^3 p$.

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

We now show the work and message complexity of algorithm $\text{DOALL}'_{\varepsilon}$, which is the main result of this section.

Theorem 5.28 There exists a set of permutations Ψ and a constant integer $\beta > 0$ such that algorithm $\text{DOALL}'_{\varepsilon}$ solves the *Do-All*$_{\mathcal{A}_S}(n, p, f)$ problem with work $S = O(n + p \cdot \min\{f + 1, \log^3 p\}$ and message complexity $M = O(fp^{\varepsilon} + p\min\{f + 1, \log p\})$.

Proof: We consider three cases:

Case 1: If the number of failures f during the execution of DGMY (recall that we execute the algorithm up to step $a \cdot (n/p + \log^3 p))$ is not greater than $\log^3 p$ then by the early-stopping property of algorithm DGMY, all non-faulty processors terminate by the end of this execution of DGMY. Work performed by the algorithm is $O(n + (f + 1)p)$ and the message complexity is $O(fp^{\varepsilon} + p\min\{f + 1, \log p\})$. This follows from the results in [28] and [44].

Case 2: If the number of failures f during the execution of DGMY is greater than $\log^3 p$ and some processor terminates in epoch -1, then by correctness of algorithm DGMY all tasks are performed, thus we stop counting work and communication and apply analysis as in previous case.

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

The analysis of the remaining epochs, starting from epoch 0, is done similarly as in Theorem 5.26. The only difference in the analysis is that we use one more epoch (epoch 0), in which the message complexity of every gossip stage is $O(p)$, if $f \le p/\log^2 p$ (per Theorem 5.27). Notice that the total number of phases is still $O(\log^2 p)$, as used in the proof of Theorem 5.26 (but constant may differ from the original). Hence the choice of set Ψ is the same as in the proof of Theorem 5.26, as well as the conditions for an integer constant $\beta > 0$, where $\beta \log p + 1$ is the number of phases in one epoch (only the constants hidden in asymptotic notation may differ, and this may increase the constant β with respect to the original one). The analysis for the general case where $f < p$ is the same as in the proof of Theorem 5.26. Therefore, we focus on executions $\xi \in \mathcal{E}(\text{DOALL}'_{\varepsilon}, \mathcal{A}_S)$ such that $\|\xi|_{\mathcal{A}_S}\| \le f \le p/\log^2 p$. We have $|V_i(\xi)| \ge p - p/\log^2 p$ for every phase i in epoch 0, and consequently the number of phases in epoch 0 sufficient to perform all the tasks, which (by the proof of Theorem 5.26 means performing work $O(n + p\log^3 p)$) is

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

Assuring that the constant hidden in the above $O(\log p)$ notation must be less than β is an additional condition for $\beta > 0$ (β must also satisfy the conditions in the proof of Theorem 5.26). This condition proves, that for every execution $\xi \in \mathcal{E}(\text{DOALL}'_{\varepsilon}, \mathcal{A}_S)$ such that $\|\xi|_{\mathcal{A}_S}\| \le f \le p/\log^2 p$, there exist a set of permutations Ψ and a constant $\beta > 0$, such that algorithm $\text{DOALL}'_{\varepsilon}$ terminates by the end of epoch 0, and by the property of algorithm $\text{GOSSIP}'_{\varepsilon/3}$, the total number of messages sent is $O(p \cdot \log p) = O(p \min\{f+1, \log p\})$, since $f > \log^3 p$ and $f \le p/\log^2 p$.

The thesis of the theorem follows from Theorem 5.26 and the three cases. $\square$

Chapter 6

Shared-Memory: Write-All with Crashes

We present failure-sensitive bounds on work for the *Write-All*$_{\mathcal{A}_S}(n, p, f)$ and r-*Write-All*$_{\mathcal{A}_S}(n, p, f)$ problems with synchronous processors, for $1 \leq f < p$, in Section 6.1. In Section 6.2 we are concerned with bounding the memory access concurrency. Kanellakis and Shvartsman [68] showed that in the presence of processor crashes, the work of any (deterministic) *Write-All* algorithm must be quadratic if processors are not allowed to access certain memory cells concurrently; specifically the showed a lower bound of $\Omega(p \cdot n)$ work for CREW (concurrent-read, exclusive-write) machines. Hence, in the presence of crashes and in the absence of concurrency, parallel computation can be extremely inefficient. However, this is not surprising, since redundancy is necessary for achieving fault-tolerance and concurrent memory access provides redundancy. Therefore, since concurrency must be allowed in order to achieve fault-tolerance and efficiency, it is interesting to understand whether concurrent memory access can be controlled in the presence of failures and at what expense on the complexity of algorithms.

6.1 Failure-Sensitive Bounds

In this section we give a new refined analysis of the most work-efficient known algorithm for the shared-memory model, algorithm W [67]. We also establish the complexity results for the iterative *Write-All* and for simulations of synchronous parallel algorithms on crash-prone processors. As in Section 5.1, our analysis is obtained by combining the results derived under the assumption of perfect knowledge for tolerating failures and the cost of achieving perfect load balancing, derived from the structure of the algorithm.

Algorithm W solves *Write-All*$_{\mathcal{A}_S}(n, p, f)$ in the shared-memory model under synchronous crash-prone processors. In [67] it was shown that the work of the algorithm is $O(n + p \log n \log p / \log\log p)$ for $p \leq n$. Observe that this bound does not include f, the number of crashes.

6.1.1 Description of Algorithm W

We now give a brief description of the algorithm but to avoid a complete restatement, we refer the reader to [68]. Algorithm W is structured as a parallel *loop* through four phases: (W1) a failure detecting phase, (W2) a load rescheduling phase, (W3) a work phase, and (W4) a phase that estimates the progress of the computation (the remaining work) and that controls the parallel loop. These phases use full binary trees with $O(n)$ leaves. The processors traverse the binary trees top-down or bottom-up according to the phase. Each such traversal takes $O(\log n)$ time (the height of a tree). For a single processor, each iteration of the loop is called a *block-step*; since there are four phases with at most one tree traversal per phase, each block-step takes $O(\log n)$ time.

In algorithm W the trees stored in shared memory serve as the gathering places for global information about the number of active processors, remaining tasks and load-balancing. Given the full details of the algorithm, it is not difficult to see that by traversing these trees synchronously, processors obtain the information that would be available from the oracle $\mathcal{O}$ in the algorithm of Figure 1, in Section 4.1. Specifically, phase W1 provides to the processors an (under)estimate on the number of operational processors and phase W4 an (over)estimate on the number of remaining tasks. This information is put together in phase W2 where the remaining tasks are allocated a balanced number of processors. The binary tree used in phase W2 to implement load-balancing and phase W3 to assess the remaining work is called the *progress tree*.

Here we use the parameterized version of the algorithm with $p \leq n$ and where the progress tree has $u = \max\{p, n/\log n\}$ leaves. The "*Do-All* tasks" are associated with the leaves of this tree, with n/u tasks per leaf. Note that each block-step still takes time $O(\log n)$.

6.1.2 Complexity Analysis

We now give the work analysis. We charge each processor for each block step it starts, regardless of whether or not the processor completes it or crashes.

Lemma 6.1 [68] The number of block-steps required by any execution of algorithm W with $f < p$ processors crashes is

$$B = O\left(u + p\frac{\log p}{\log\log p}\right).$$

Lemma 6.2 The number of block-steps required by any execution of algorithm W with $f \leq \frac{p}{\log p}$ processors crashes is

$$B = O\left(u + p\log_{\frac{p}{f}} p\right).$$

Proof: It is not difficult to see, that the processor block-steps are equivalent to the processor steps under the assumption of perfect knowledge. Hence, the proof is the same as the proof of Lemma 4.2. $\square$

Theorem 6.3 Algorithm W solves *Write-All*$_{\mathcal{A}_S}(n,p,f)$ using work

$$S = O\left(n + p\log n \frac{\log p}{\log(p/f)}\right) \text{ when } f \leq \frac{p}{\log p}, \text{ and}$$

$$S = O\left(n + p\log n \frac{\log p}{\log\log p}\right) \text{ when } \frac{p}{\log p} < f < p.$$

Proof: We consider the following two cases:

Case 1: $p < \frac{n}{\log n}$. Here the number of leaves in the progress tree is $u = n/\log n$ and in the work phase W3 each processor performs $n/u = \log n$ tasks. The cost of a single block-step is $C_1 = O(\log n)$ since each of the four phases takes at most $\log n$ time. We consider two subcases:

$(1a)$ $f \leq \frac{p}{\log p}$. Per Lemma 6.2, the number of blocks-steps B_{1a} for this case is:

$$B_{1a} = O\left(u + p\frac{\log p}{\log \frac{p}{f}}\right) = O\left(\frac{n}{\log n} + p\frac{\log p}{\log \frac{p}{f}}\right).$$

Therefore,

$$S_{1a} = B_{1a} \cdot C_1 = O\left(\frac{n}{\log n} + p\frac{\log p}{\log \frac{p}{f}}\right) \cdot O(\log n) = O\left(n + p\log n\frac{\log p}{\log \frac{p}{f}}\right).$$

$(1b)$ $f > \frac{p}{\log p}$. Per Lemma 6.1, the number of block-steps B_{1b} for this case is:

$$B_{1b} = O\left(u + p\frac{\log p}{\log\log p}\right) = O\left(\frac{n}{\log n} + p\frac{\log p}{\log\log p}\right).$$

Therefore,

$$S_{1b} = B_{1b} \cdot C_1 = O\left(\frac{n}{\log n} + p\frac{\log p}{\log\log p}\right) \cdot O(\log n) = O\left(n + p\log n\frac{\log p}{\log\log p}\right).$$

Case 2: $\frac{n}{\log n} \leq p \leq n$. Here the number of leaves in the progress tree is $u = p$ and in the work phase W3 each processor performs $\lceil n/p \rceil = O(\log n)$ tasks. Thus the cost of a single block-step is $C_2 = O(\log n)$. We again consider two subcases:

$(2a)$ $f \leq \frac{p}{\log p}$. Per Lemma 6.2, the number of block-steps B_{2a} for this case is:

$$B_{2a} = O\left(u + p\frac{\log p}{\log \frac{p}{f}}\right) = O\left(p + p\frac{\log p}{\log \frac{p}{f}}\right) = O\left(p\frac{\log p}{\log \frac{p}{f}}\right).$$

Therefore,

$$S_{2a} = B_{2a} \cdot C_2 = O\left(p\frac{\log p}{\log \frac{p}{f}}\right) \cdot O(\log n) = O\left(p \log n\frac{\log p}{\log \frac{p}{f}}\right).$$

$(2b)$ $f > \frac{p}{\log p}$. Per Lemma 6.1, the number of block-steps B_{2b} for this case is:

$$B_{2b} = O\left(p + p\frac{\log p}{\log\log p}\right) = O\left(p\frac{\log p}{\log\log p}\right).$$

Therefore,

$$S_{2b} = B_{2b} \cdot C_2 = O\left(p\frac{\log p}{\log\log p}\right) \cdot O(\log n) = O\left(p\log n\frac{\log p}{\log\log p}\right).$$

Combining Case 1 and Case 2 we obtain the desired result for $1 \leq p \leq n$. □

6.1.3 Iterative Write-All and Parallel Algorithm Simulations

We now consider the complexity of shared-memory synchronous r-*Write-All*$_{\mathcal{A}_S}(n, p, f)$ and of simulations of parallel algorithms on crash-prone processors.

Theorem 6.4 The r-*Write-All*$_{\mathcal{A}_S}(n, p, f)$ problem can be solved on p synchronous crash-prone processors with work

$$S = O\left(r \cdot \left(n + p\log n\frac{\log p}{\log(pr/f)}\right)\right) \text{ when } f \leq \frac{pr}{\log p}, \text{ and}$$
$$S = O\left(r \cdot \left(n + p\log n\frac{\log p}{\log\log p}\right)\right) \text{ when } \frac{pr}{\log p} < f < p.$$

Proof: The iterative *Write-All* can be solved by running algorithm W on r instances of size n in sequence. We call this algorithm W*. To analyze the efficiency of W* we use the same approach as in the proof of Theorem 4.11. In the current context we base our work complexity arguments on the result of Theorem 6.3. □

The above result on iterative *Write-All* leads to the following result for PRAM simulations:

Theorem 6.5 Any synchronous n-processor, r-time shared-memory parallel algorithm (PRAM) can be simulated on p crash-prone synchronous processors with work

$$S = O\left(r \cdot \left(n + p \log n \frac{\log p}{\log(pr/f)}\right)\right) \text{ when } f \leq \frac{pr}{\log p}, \text{ and}$$

$$S = O\left(r \cdot \left(n + p \log n \frac{\log p}{\log \log p}\right)\right) \text{ when } \frac{pr}{\log p} < f < p.$$

Proof: The complexity of simulating a single parallel step of n ideal processors on p crash-prone processors does not exceed the complexity of solving a single *Write-All*$_{\mathcal{A}_S}(n, p, f)$ instance [72, 104]. The result then follows from Theorem 6.4. □

This last result shows a failure-sensitive improvement over the previously known bounds of $O\left(r \cdot (n + p \log n \log p / \log \log p)\right)$ for deterministic parallel algorithm simulations on crash-prone processors [104].

6.2 Failure-Sensitive Bounds for Controlled Memory Access Concurrency

In this section we derive failure-sensitive bounds on work for the *Write-All* and *iterative Write-All* problems in the setting where memory access concurrency must be controlled. In particular, we give a new failure-sensitive analysis of algorithm KMS [66] (the only algorithm for *Write-All* in this setting) and we refine its range of optimality. We then use the algorithm to establish new failure-sensitive bounds on work for the *iterative Write-All* problem for synchronous shared-memory systems, while simultaneously bounding memory access concurrency. This yields tighter bounds on work (vs. [66]) for simulations of parallel algorithms on crash-prone processors with bounded memory access concurrency. Our analysis is performed by separately assessing the cost of tolerating failures derived from the results under the assumption of perfect knowledge and the cost of implementing perfect load balancing, derived from the structure of the algorithm.

6.2.1 Description of Algorithm KMS

We now give a brief description of the algorithm but to avoid a complete restatement, we refer the reader to [66]. The algorithm consists of two layers, where the top layer provides the overall control structure for solving *Write-All* and the bottom layer is responsible for controlling memory access concurrency.

The top layer control structure is based on algorithm W [67]: It consists of the main loop that iterates through fours phases (phases W1,W2,W3 and W4; see Section 6.1.1) until the *Write-All* problem is solved. The algorithm uses two complete binary trees: the *processor enumeration tree* with p leaves (used in phase W1 to detect failed processors and renumber the processors compactly) and the *progress tree* (used in phase W2 to implement load-balancing and in phase W3 to assess the remaining work) with h leaves ($1 \leq h \leq n$), where a cluster of n/h elements of the *Write-All* array (or the "*Do-All*" tasks) are associated with each leaf.

The bottom layer provides specific access routines for reading from, and writing to, the shared memory; it uses two data structures represented as binary trees: (1) The *processor priority tree* (PPT) coordinates access to memory by determining which processors are allowed to read or write each shared location that has to be accessed concurrently by more than one processor. The nodes of the tree are associated with processors based on a processing numbering. Priorities are assigned to the processors according to the tree levels: the root has the highest priority and priority decrease with each successive level. In the top layer, processors traverse the progress and enumeration trees in a bottom-up fashion. At each intermediate node of a tree two PPTs need to be combined into one as the processors that come up form the children of the node "meet" at the parent. This involves *compacting* and *merging* the PPTs. PPTs are compacted to eliminate "certifiably" faulty processors. Two PPTs are merged by having the processors of the left PPT appended to the tree formed by the processors of the right one (see [66] for details). (2) The *broadcast tree* is used to disseminate values among readers and writers. The use of broadcast trees in conjunction with priority trees serves to bound read and write concurrency.

Algorithm CR/W is the main algorithm of the bottom layer that uses the above structures to control memory access concurrency for individual reads and writes. Specifically, it implements broadcast (using the broadcast tree) for processors within different levels of a PPT and allows processors to write to a shared location L only if processors at higher levels haven't done so. Communication between processors in a PPT takes place through a shared memory array, call it B, where the processors communicate based on their positions in the PPT. $B[k]$ stores values read by the kth processor of the PPT. Each processor on levels $0, \ldots, i-1$ is associated with exactly one processor on each of the levels i and lower. Specifically, the jth processor of the PPT broadcasts to the jth processor of each level below its own (in a left-to-right numbering within each level). The algorithm proceeds in $\lfloor \log p \rfloor + 1$ iterations that correspond to the PPT levels. At iteration i, each processor of level i reads its B location. If this location has not been updated, then the processor reads L directly. Since each full PPT level has one more processor than all the levels above it combined (PPT is a binary tree), there may be at least one processor on each level that reads L directly since no processor at a higher level is assigned to it (for a full level, this processor is the rightmost one, or the root itself for level 0). As long as there are no failures this is the only direct access to L. Concurrent accesses can occur only in the presence of failures. In such a case several processors on the same level may fail to receive values from processors at higher levels, in which case they concurrently read L directly. A processor reading L directly checks whether it contains the value to be written, then writes to it if it does not. Whenever processors update L they write the new value for L as well as the index of the level that effected the write. If a processor k accesses L and determines that L has the correct value, and if the failed processor ℓ that should have broadcast to k is at or below the level that effected the write, then k assumes the position of processor ℓ in the PPT. This effectively moves failed processors toward the leaves of the PPT. Failed processors are moved downwards only if they are not above the level that effects the write – processors above this level are eliminated by PPT compaction that takes place at the end of each run of CR/W.

Algorithm CR/W combines a read with a write. However, when the processors of a PPT need to read a common location but no write is involved, two simpler algorithms are used: Algorithm CR1 which is used for bottom-up traversals and algorithm CR2 which is used for top-down traversals. Algorithm CR1 is similar to CR/W but includes no write step. This algorithm is simpler than CR/W in that the processors that are found to have failed are pushed toward the bottom of the PPT independent of their level. Algorithm CR2 uses a simple top-down broadcast through the PPT. Starting with the root each processor broadcasts to its two children; if a processor fails then its two children read T directly. Thus the processors of level i broadcast only to processors of level $i+1$. Unlike CR1, no processor movement takes place.

From the description of algorithms CR/W, CR1, and CR2 it follows that each takes time $O(\log p)$.

We now describe how algorithm KMS integrates algorithms CR/W, CR1, CR2, and PPT merging and compaction within its four phases.

Phase 1: Processors begin this phase by forming single-processor PPTs. The objective is to write to each internal node of the enumeration tree the sum of the values stored at its two children. Algorithm CR/W is used to store the new value, the size of the PPT and the index of the level that completed the write. Then all PPTs are compacted. In order to merge PPTs the processors use algorithm CR1 to read the data stored at the enumeration tree node that is the sibling of the node they just updated. Then PPTs are merged. At this point the processors of the merged PPTs know the value they need to write at the next level of the enumeration tree. This value is the sum of the value written by CR/W and the value read by CR1. Therefore one call to each of CR/W and CR1 is needed for each level of the enumeration tree.

Phase 2: This phase involves no concurrent writes. Processors traverse top-down the progress tree to allocate themselves to the unvisited leaves. The only global information needed at each level is the values stored at the two children of the current node of the progress tree. Two calls to CR2 are used to read these values, one for each child. Using this information the processors of a PPT compute locally whether they need to go left or right based on their identifiers. Here each PPT must be split in two. If a PPT has k processors of which k' need to go left and the remaining $k - k'$ need to go right, then by convention the first k' processors of the PPT form the PPT of the left child and the remaining $k - k'$ processors form the PPT of the right child. No compaction or merging is done in this phase.

Phase 3: Processors form PPTs based on the information they gathered during *Phase 2* and proceed to write 1 to the n/h locations that correspond to the leaf they reached. At this point, processors decide whether they need to use algorithm CR/W, followed by compaction for each of these writes. This is done locally by each processor: at the beginning of this phase, the processors have consistent information on the number of unvisited leaves, call it u, and the number of available processors, call it a (this is the information they used to allocate themselves at the leaves they reached by the end of *Phase 2*). When $u > a$, it is guaranteed (see [66]) that there is at most one processor per leaf, and therefore the processors do not use CR/W and compaction. Instead the processors go sequentially through the cluster of n/h elements at the leaf they reached and simply write to each element. When $u \leq a$, several processors may be allocated to the same leaf and the processors use algorithm CR/W followed by compaction to perform each write in the cluster. In any case, no merging is involved.

Phase 4: This phase initially uses the PPTs that resulted at the end of *Phase 3*. The task to be performed is similar to that of *Phase 1*. As before, algorithm CR/W is used for writing followed by compaction and one call to algorithm CR1, after which the PPTs are merged.

We now state previously known results [66] for algorithm KMS and for simulations using this algorithm.

Theorem 6.6 [66] Algorithm KMS solves the *Write-All*$_{\mathcal{A}_S}(n, p, f)$ problem with work $S = O\left(n + p\log^2 n \log^2 p/\log\log n\right)$, write concurrency $\omega \leq f$, and read concurrency $\rho \leq 7\,f\log n$.

Theorem 6.7 [66] Any n-processor, r-time exclusive-read, exclusive-write parallel algorithm (EREW PRAM) can be simulated on a p synchronous crash-prone processors with work $S = O\left(r \cdot \left(n + p\log^2 p \log^2 n/\log\log n\right)\right)$, with write concurrency $\omega \leq f$, and the read concurrency $\rho \leq 7f\log n$

These prior results do not show how the work depends on the number of processor crashes.

6.2.2 Complexity Analysis

We now give a new, failure-sensitive, analysis of algorithm KMS, based on the results obtained for *Do-All* under the assumption of perfect knowledge.

In the analysis we use the parameterized version of algorithm KMS with $p \leq n$ and where the progress tree has $u = \max\{p, n/\log n \log p\}$ leaves. The array elements are associated with the leaves of this tree, with n/u array elements per leaf. Henceforth we use KMS to denote this parameterized algorithm.

For an execution of algorithm KMS, we define u_i to be the number of unvisited leaves of the progress tree ($u_i \leq u$), and p_i to be the number of non-faulty processors ($p_i \leq p$), at the start of the i-th iteration of the main loop. We define σ_1 to be the time required for a processor to complete one iteration of the main loop when $p_i < u_i$. We define σ_2 to be the time required for a processor to complete one iteration of the main loop when $p_i \geq u_i$. We define a *block-step* to be the execution by one processor of the body of the main loop.

Lemma 6.8 The work required by algorithm KMS to solve the *Write-All*$_{\mathcal{A}_S}(n, p, f)$ problem is $S = O\left(\sigma_1 \cdot u + \sigma_2 \cdot \dfrac{p \log p}{\log \log p}\right)$.

Proof: We consider two cases.

Case 1: Consider *all* iterations i in which $p_i < u_i$. In this case the number of block-steps is $O(u)$ since no more than one processor is assigned to each leaf of the progress tree. Then, using the definition of σ_1, the work of algorithm KMS in this case is $O(\sigma_1 \cdot u)$.

Case 2: We now account for *all* iterations in which $p_i \geq u_i$. In this case the number of block-steps is $O(p \frac{\log p}{\log \log p})$. Given the load-balancing properties of algorithm KMS, this follows directly from the case analysis of Theorem 3.1 [50], where Case 2 considers the work of perfect load-balancing iterative algorithms when $p_i > u_i$. (The simpler subcase of $p_i = u_i$ is dealt similarly.) Then, using the definition of σ_2, the work of algorithm KMS in this case is $O(\sigma_2 \cdot p\frac{\log p}{\log\log p})$.

Combining the two cases yields the result. □

Note that in the above lemma, work is not expressed as a function of f, the number of processor crashes. In the next lemma, we give work as a function of f, for $f \leq p/\log p$. The proof of the lemma is based on the proof of Lemma 4.2

Lemma 6.9 The work required by algorithm KMS to solve the *Write-All*$_{\mathcal{A}_S}(n, p, f)$ problem when $f \leq \frac{p}{\log p}$ is $S = O\left(\sigma_1 \cdot (u+p) + \sigma_2 \cdot \dfrac{p \log u}{\log(p/f)}\right)$.

Proof: Let u' be the number of unvisited leaves of the progress tree (recall that the tree has u leaves with n/u array elements assigned to each leaf). Let Δf denote the number of processor crashes within a particular iteration of an execution of the algorithm. Δf is, in general, different for each iteration, though the sum of these for all iterations cannot exceed f. We set $b = b(p, f) = p/(2f)$, and we define $S(u', p, f)$, where $u' \leq u$, to be the work required to solve *Write-All*$_{\mathcal{A}_S}(u' \cdot n/u, p, f)$. We show that for all u', p and f, $S(u', p, f)$ is no more than $\sigma_1(p + u') + 3\sigma_2 p + \sigma_2 p \log_{p/(2f)} u'$. The proof proceeds by induction on u' (following our approach in Lemma 4.2).

Base Case: Observe that when $u' = 1$ and $p \geq 1$ (hence $p \geq u'$), $S(u', p, f) \leq \sigma_2 p \leq \sigma_1(p + u') + 3\sigma_2 p + \sigma_2 p \log_b u'$, for all p and f, as desired.

Inductive Hypothesis: Assume that we have proved the result for all $u' < \hat{u}$ and all p and f.

Inductive Step: Consider $u' = \hat{u}$. We investigate two cases:

Case 1: $p < \hat{u}$. In this case each processor is assigned to a unique unvisited leaf (this follows from the load-balancing properties of algorithm KMS), hence

$$S(\hat{u}, p, f) \leq \sigma_1 p + \max_{0 \leq \Delta f \leq f} S(\hat{u} - p + \Delta f, p - \Delta f, f - \Delta f).$$

As $p - \Delta f > 0$, $\hat{u} - p + \Delta f < \hat{u}$ and, by the induction hypothesis,

$$S(\hat{u}, p, f) \leq \sigma_1 p + \max_{0 \leq \Delta f \leq f} \Big[\sigma_1(p - \Delta_f + \hat{u} - p + \Delta_f) + 3\sigma_2(p - \Delta f)$$
$$+ \sigma_2(p - \Delta f) \log_{b(p-\Delta f, f-\Delta f)}(\hat{u} - p + \Delta f)\Big].$$

Now, $b(p - \Delta f, f - \Delta f) \geq b(p, f)$, so that

$$S(\hat{u}, p, f) \leq \sigma_1(p + \hat{u}) + 3\sigma_2 p + \sigma_2 p \log_{b(p,f)} \hat{u},$$

as desired.

Case 2: $p \geq \hat{u}$. In this case, by assumption we have

$$S(\hat{u}, p, f) \leq \sigma_2 p + \max_{0 \leq \Delta f \leq f} S(\gamma\hat{u}, p - \Delta f, f - \Delta f),$$

where $\gamma = \gamma(\hat{u}, p, \Delta f)$ is the ratio of the number of the remaining unvisited leaves to $\hat{u}$ ($0 \leq \gamma < 1$). Let $\phi = \Delta f/p \leq f/p < 1$, the fraction of processors which fail during this iteration; then $\phi/2 < \gamma < 2\phi$ (see proof of Lemma 4.2). Then,

$$S(\hat{u}, p, f) \leq \sigma_2 p + \max_{\phi \in [0, f/p]} S(\gamma\hat{u}, (1 - \phi)p, f - \phi p).$$

As $\gamma\hat{u} < \hat{u}$, we may apply the induction hypothesis:

$$S(\hat{u}, p, f) \leq \sigma_2 p + \max_{\phi \in [0, f/p]} \Big[\sigma_1(\gamma\hat{u} + (1 - \phi)p) + 3\sigma_2(1 - \phi)p$$
$$+ \sigma_2(1 - \phi)p \log_{b'}(\gamma\hat{u})\Big],$$

where $b' = b(p - \phi p, f - \phi p)$. As above, $b' \geq b(p, f)$, so that

$$S(\hat{u}, p, f) \leq \sigma_2 p + \max_{\phi \in [0, f/p]} \Big[\sigma_1(\gamma\hat{u} + (1 - \phi)p) + 3\sigma_2(1 - \phi)p$$
$$+ \sigma_2(1 - \phi)p \log_{b(p,f)}(\gamma\hat{u})\Big].$$

To complete the proof, it suffices to show that for all $\phi \in [0, f/p]$,

$$\sigma_1 \phi p + 2\sigma_2 p + \sigma_2 p \log_{b(p,f)} \hat{u} - (1 - \phi)\sigma_2 p \log_{b(p,f)}(\gamma\hat{u}) \geq 3\sigma_2(1 - \phi)p - \sigma_1\hat{u}(1 - \gamma).$$

Upper bounding $3\sigma_2(1-\phi)p - \sigma_1\hat{u}(1-\gamma)$ with $3\sigma_2(1-\phi)p$, removing $\sigma_1\phi p$ from the left hand side, and dividing through by $\sigma_2 p$, it is sufficient to show that

$$2 + \log_{b(p,f)} \hat{u} - (1-\phi)\log_{b(p,f)}(\gamma\hat{u}) \geq 3(1-\phi),$$

or, equivalently,

$$\log_{b(p,f)} \hat{u} - (1-\phi)\log_{b(p,f)}(\gamma\hat{u}) \geq 1 - 3\phi.$$

We now focus on the left hand side of the above equation:

$$\log_{b(p,f)} \hat{u} - (1-\phi)\left[\log_{b(p,f)} \gamma + \log_{b(p,f)} \hat{u}\right] = \phi \log_{b(p,f)} \hat{u} + (1-\phi)\log_{b(p,f)} \gamma^{-1}.$$

Since $f \leq p/\log p$, for any $p \geq 16$ we have that $p/(2f) > 2$. Observe that,

$$\phi \log_{b(p,f)} \hat{u} + (1-\phi)\log_{b(p,f)} \gamma^{-1} \geq (1-\phi)\log_{b(p,f)} \gamma^{-1}$$

since $\hat{u} \geq p/f > p/(2f)$. (Note that if $\hat{u} < p/f$, then all leaves are visited in this iteration.) Recall that $\gamma^{-1} \geq (2\phi)^{-1}$ and $\phi < f/p$. Therefore,

$$(1-\phi)\log_{b(p,f)} \gamma^{-1} \geq (1-\phi)\log_{b(p,f)}(2\phi)^{-1} \geq 1 - 3\phi.$$

Evidently,

$$S = O\left(\sigma_1 \cdot (u' + p) + \sigma_2 \cdot \frac{p \log u'}{\log(p/f)}\right) = O\left(\sigma_1 \cdot (u + p) + \sigma_2 \cdot \frac{p \log u}{\log(p/f)}\right),$$

as desired. □

Lemma 6.10 Algorithm KMS solves the *Write-All*$_{\mathcal{A}_S}(n,p,f)$ problem with work

$$S = O\left(\sigma_1 \cdot (u+p) + \sigma_2 \cdot p\frac{\log n}{\log(p/f)}\right) \text{ when } f \leq \frac{p}{\log p}, \text{ and}$$

$$S = O\left(\sigma_1 \cdot (u+p) + \sigma_2 \cdot p\frac{\log n}{\log\log p}\right) \text{ when } \frac{p}{\log p} < f < p.$$

Proof: We first record that $u < u+p$, $\log p \leq \log n$ and $\log u \leq \log n$. Then the result follows by combining Lemmas 6.8 and 6.9. □

The above result shows the cost (work) of tolerating failures, while the cost of implementing load-balancing is hidden in σ_1 and σ_2. We now compute the cost of implementing load-balancing by algorithm KMS (that is, compute the values of σ_1 and σ_2).

Lemma 6.11 For algorithm KMS, $\sigma_1 = O(\log n \log p)$ and $\sigma_2 = O(\log n \log^2 p)$.

Proof: We consider the following two cases.

Case 1: $p < \frac{n}{\log n \log p}$. Here the number of leaves in the progress tree is $u = n/\log n \log p$ and in *Phase 3* each processor writes to $n/u = \log n \log p$ array elements. The time required to traverse the enumeration and progress trees is $O(\log n \log p)$ and the execution of CR/W takes $O(\log p)$ time.

For the iteration i when $u_i \geq p_i$, algorithm CR/W is not used in *Phase 3* and therefore the time to update a leaf is $O(\log n \log p)$ (the number of elements). Therefore, $\sigma_1 = O(\log n \log p) + O(\log n \log p) = O(\log n \log p)$ (the time to reach a leaf plus the time to update a leaf).

For the iteration i when $u_i < p_i$, algorithm CR/W is used in *Phase 3*. In the worst case, all processors could be allocated to the same leaf (e.g., when there is only one unvisited leaf left) and hence, $\log p$ time must be spent at each element of the leaf. Since there are $\log n \log p$ elements per leaf the worst case time to update a leaf is $O(\log n \log^2 p)$. Hence, $\sigma_2 = O(\log n \log p) + O(\log n \log^2 p) = O(\log n \log^2 p)$.

Case 2: $\frac{n}{\log n \log p} \leq p \leq n$. Here the number of leaves in the progress tree is $u = p$ and in *Phase 3* each processor writes to $n/p = O(\log n \log p)$ array elements. Then the bounds on σ_1 and σ_2 are obtained similarly to Case 1. $\square$

We now state and prove our main result for algorithm KMS.

Theorem 6.12 Algorithm KMS solves the *Write-All*$_{\mathcal{A}_S}(n, p, f)$ problem with write concurrency $\omega \leq f$, read concurrency $\rho \leq 7\, f \log n$ and work

$$S = O\left(n + p \log^2 n \frac{\log^2 p}{\log(p/f)}\right) \text{ when } f \leq \frac{p}{\log p}, \text{ and}$$

$$S = O\left(n + p\log^2 n \frac{\log^2 p}{\log\log p}\right) \text{ when } \frac{p}{\log p} < f < p.$$

Proof: The bounds on ω and ρ are obtained from Theorem 6.6 (see [66]). We now show the bounds on S. The bounds are derived by combining the cost of tolerating failures and the cost implementing load-balancing. We consider two cases:

Case 1: $p < \frac{n}{\log n \log p}$. Here the number of leaves in the progress tree is $u = n/\log n \log p$. Combining Lemmas 6.10 and 6.11 we get $S = O(\sigma_1 \cdot (u + p) + \sigma_2 \cdot p \log n/\log(p/f)) = O((\log n \log p) \cdot n/(\log n \log p) + (\log n \log^2 p) \cdot p \log n/\log(p/f)) = O(n + p\log^2 n \log^2 p/\log(p/f))$ when $f \leq p/\log p$ and similarly $S = O(n + p\log^2 n \log^2 p/\log\log p)$ when $f > p/\log p$.

Case 2: $\frac{n}{\log n \log p} \leq p \leq n$. Here the number of leaves in the progress tree is $u = p$. Combining Lemmas 6.10 and 6.11 we have $S = O(\sigma_1 \cdot (u+p) + \sigma_2 \cdot p \log n/\log(p/f)) = O((\log n \log p) \cdot p + (\log n \log^2 p) \cdot p \log n/\log(p/f)) = O(p\log^2 n \log^2 p/\log(p/f))$ when $f \leq p/\log p$ and similarly $S = O(p\log^2 n \log^2 p/\log\log p)$ when $f > p/\log p$.

The result is obtained by combining Case 1 and Case 2. □

This analysis establishes the following processor ranges for which algorithm KMS becomes optimal.

Corollary 6.13 Algorithms KMS is work-optimal if $p = O(n\log(n/f)/\log^4 n)$, when $f \leq p/\log p$, and if $p = O(n\log\log n/\log^4 n))$, when $f > p/\log p$.

Theorem 6.6 teaches that algorithm KMS becomes optimal if $p = O(n\log\log n/\log^4 n)$, for all $f < p$. Corollary 6.13 shows that our failure-sensitive analysis extends the range of optimality of the algorithm when $f \leq p/\log p$.

6.2.3 Iterative Write-All and Parallel Algorithm Simulations

Using algorithm KMS and its new analysis, we obtain new failure-sensitive bounds for the *iterative Write-All* problem with controlled read and write memory access concurrency.

Theorem 6.14 The r-*Write-All*$_{\mathcal{A}_S}(n, p, f)$ problem can be solved on p synchronous crash-prone processors with write concurrency $\omega \le f$, read concurrency $\rho \le f \log n$ and work

$$S = O\left(r \cdot \left(n + p\log^2 n \frac{\log^2 p}{\log(pr/f)}\right)\right) \text{ when } f \le \frac{pr}{\log p}, \text{ and}$$
$$S = O\left(r \cdot \left(n + p\log^2 n \frac{\log^2 p}{\log\log p}\right)\right) \text{ when } \frac{pr}{\log p} < f < p.$$

Proof: We solve r-*Write-All*$_{\mathcal{A}_S}(n, p, f)$ by running algorithm KMS r times, once for each *Write-All* instance. We enumerate the r instances of *Write-All* using numbers $1, \ldots, r$, and we refer to instance i as the *round* i. For round i, let p_i be the number of active processors at the beginning of the round and f_i be the number of crashes during the round. Note that $p_1 = p$, and that $p_i \le p$.

We first establish the bounds on the memory access concurrency. Let ω_i and ρ_i be the write and read memory access concurrency accrued in round i, respectively. Then, $\omega = \sum_{i=1}^{r} \omega_i$ and $\rho = \sum_{i=1}^{r} \rho_i$. Using Theorem 6.12 for each round, we have that $\omega_i \le f_i$ and $\rho_i \le f_i \log n$. Therefore,

$$\omega = \sum_{i=1}^{r} \omega_i \le \sum_{i=1}^{r} f_i = f, \text{ and } \rho = \sum_{i=1}^{r} \rho_i \le \log n \sum_{i=1}^{r} f_i = f \log n,$$

as desired.

Observe that the choice of each f_i does not affect the bounds on the memory access concurrency. However, in order to establish the bounds on work we need to determine the values of the f_is that maximize the overall work of r-*Write-All*$_{\mathcal{A}_S}(n, p, f)$. The work analysis is done as in the proof of Theorem 4.11. In the current context we base our work complexity arguments on the result of Theorem 6.12. $\square$

Theorem 6.14 enables us to obtain a tighter bound on work when algorithm KMS is iteratively used to obtain efficient parallel algorithm simulations on crash-prone processors (as opposed to the bound of Theorem 6.7).

Theorem 6.15 Any n processor, r-time exclusive-read, exclusive-write parallel (EREW PRAM) algorithm can be simulated on p synchronous crash-prone processors with work

$$S = O\left(r \cdot \left(n + p\log^2 n \frac{\log^2 p}{\log(pr/f)}\right)\right) \text{ when } f \leq \frac{pr}{\log p}, \text{ and}$$

$$S = O\left(r \cdot \left(n + p\log^2 n \frac{\log^2 p}{\log\log p}\right)\right) \text{ when } \frac{pr}{\log p} < f < p,$$

so that the write concurrency of the simulation is $\omega \leq f$ and the read concurrency is $\rho \leq 7f\log n$.

Proof: The complexity of simulating a single parallel step of n ideal processors on p failure-prone processors does not exceed the complexity of solving a single *Write-All*$_{\mathcal{A}_S}(n, p, f)$ instance [72, 104]. The result then follows from Theorem 6.14. □

Note that this last result can be extended to other PRAM variants, such as concurrent-read, exclusive-write (CREW) and concurrent-read, concurrent-write (CRCW), however in these cases the read and write concurrency bounds depend on the actual read and write concurrency of the simulated algorithm. Another way is to convert the simulated algorithm into an equivalent EREW algorithm (using the standard PRAM conversion techniques [69]). Then, the simulation obtains the same concurrency bounds as in Theorem 6.15 at the expense of increasing the work by a logarithmic factor (the overhead is due to the cost of the conversion).

Chapter 7

Omni-Do in Partitionable Networks

In the settings where network partitions may interfere with the progress of computation, the challenge is to maintain efficiency in performing the tasks and learning the results of the tasks (solving *Omni-Do*), despite the dynamically changing group connectivity. However, no amount of algorithmic sophistication can compensate for the possibility of groups of processors or even individual processors becoming disconnected during the computation. In general, an adversary that is able to partition the network into g components will cause any task-performing algorithm to have work $\Omega(n \cdot g)$ even if each group of processors performs no more than the optimal number of $\Theta(n)$ tasks. In the extreme case where all processors are isolated from the beginning, the work of any algorithm is $\Omega(n \cdot p)$.

Even given the pessimistic lower bounds on work for partitionable networks, it is desirable to design and analyze efficient algorithmic approaches that can be shown to be better than the oblivious approach where each processor or each group performs all tasks. In Section 7.1 we extend the work of Dolev, Segala, and Shvartsman [32]. We present an asynchronous *Omni-Do* algorithm, called AX, and we show that it is optimal in terms of worst case task-oriented work, under fragmentations and merges (as opposed to the algorithm in [32] that deals only with fragmentations). The algorithm uses a group communication service [95] to provide membership and communication services.

An *Omni-Do* algorithm and its efficiency can only be partially understood through its worst case work analysis. This is because the resulting worst case bound might depend on unusual or extreme patterns of regroupings. In such cases, worst case work may not be the best way to compare the efficiency of algorithms. Hence, in Section 7.2, in order to understand better the practical implications of performing work in partitionable settings, we initiate the study of the *Omni-Do* problem as an on-line problem and we pursue *competitive analysis* [105]. In particular, we study a simple randomized algorithm, called RS, and we compare its expected task-oriented work to the task-oriented work of an "off-line" algorithm that has full knowledge of future changes in the communication medium. We show that algorithm RS is "optimally-task-oriented-work-competitive" under arbitrary patterns of regroupings, including but not limited to fragmentations and merges.

7.1 Worst Case Analysis of Omni-Do

In this section we present algorithm AX and we show that it is work-optimal under adversary $\mathcal{A}_{FM}$. We assume that initially the processors belong to a single group. The algorithm specification is done in terms of *Input/Output Automata* of Lynch and Tuttle [81, 80]. In Section 7.1.1 we give a brief introduction to Input/Output Automata. In Section 7.1.2 we present the group communication service used for providing membership and communication services. In Section 7.1.3 we define *view-graphs* that we use in the analysis. In Section 7.1.4 we describe algorithm AX and we show its correctness. Finally, in Section 7.1.5 we present the complexity analysis of the algorithm.

7.1.1 Input/Output Automata

The algorithm specification is done in terms of Input/Output automata of Lynch and Tuttle [81, 80]. Each automaton is a state machine with states and transitions between states, where actions are associated with sets of state transitions. There are input, output and internal actions. A particular action is enabled if the preconditions of that action are satisfied. Input actions are always enabled. The statements given as effects are executed as a program started in the existing state and atomically producing the next state as the result of the transition.

An execution ξ of an Input/Output automaton Aut is a finite or infinite sequence of alternating states and actions (events) of Aut starting with the initial state, i.e., $\xi = s_0, e_1, s_1, e_2, \ldots$, where s_i's are states (s_0 is the initial state) and e_i's are actions (events). We denote by *execs*(Aut) the set of all executions in Aut.

Consider an algorithm Λ that is specified in I/O automata and it solves a specific problem under an adversary $\mathcal{A}$. Then, following the notation established in Section 3.2.2, *execs*$(\Lambda) = \mathcal{E}(\Lambda, \mathcal{A})$.

7.1.2 A Group Communication Service

We assume a group communication service (GCS) with certain properties. The assumptions are basic, and they are provided by several group communication systems and specifications [23]. The service maintains group membership information and it is used to communicate information concerning the executed tasks within each group. Each processor, at each time, has a unique *view* of the membership of the group. The view includes a list of the processors that are members of the group. Views can change and may become different at different processors. The GCS provides the following primitives:

- NEWVIEW$(v)_i$: informs processor i of a new view $v = \langle id, set \rangle$, where id is the identifier of the view and set is the set of processor identifiers in the group. When a NEWVIEW$(v)_i$ primitive is invoked, we say that processor i *installs* view v.

- GPMSND(*message*)$_i$: processor i multicasts a message to the group members.

- GPMRCV(*message*)$_i$: processor i receives multicasts from other processors.

- GP1SND(*message,destination*)$_i$: processor i unicasts a message to another member of the current group.

- GP1RCV(*message*)$_i$: processor i receives unicasts from another processor.

To distinguish between the messages sent in different send events, we assume that each message sent by the application is tagged with a unique message identifier.

We assume the following safety properties on any execution ξ of an algorithm that uses GCSs:

1. A processor is always a member of its view ([23] Prop. 3.1). If NEWVIEW$(v)_i$ occurs in ξ then $i \in v.set$.

2. The view identifiers of the views that each processor installs are monotonically increasing ([23] Prop.3.2). If event NEWVIEW$(v_1)_i$ occurs in ξ before event NEWVIEW$(v_2)_i$, then $v_1.id < v_2.id$. This property implies that: (a) A processor does not install the same view twice, and (b) if two processors install the same two views, they install these views in the same order.

3. For every receive event, there exists a preceding send event of the same message ([23] Prop. 4.1). If GPMRCV$(m)_i$ (GP1RCV$(m)_i$) occurs in ξ, then there exists GPMSND$(m)_j$ (GP1SND$(m, i)_j$) earlier in execution ξ.

4. Messages are not duplicated ([23] Prop. 4.2). If GPMRCV$(m_1)_i$ (GP1RCV$(m_1)_i$) and GPMRCV$(m_2)_i$ (GP1RCV$(m_2)_i$) occur in ξ, then $m_1 \neq m_2$.

5. A message is delivered in the same view it was sent in ([23] Prop. 4.3). If processor i receives message m in view v_1 and processor j (it is possible that $i = j$) sends m in view v_2, then $v_1 = v_2$.

6. In the initial state s_0, all processors are in the initial view v_0, such that $v_0.set = \mathcal{P}$ ([23] Prop. 3.3 with [39, 88]).

We assume the following additional liveness properties on any execution ξ of an algorithm that uses GCSs (cf. [23] Section 10):

7. If a processor i sends a message m in the view v, then for each processor j in $v.set$, either j delivers m in v, or i installs another view (or i crashes).

8. If a new view event occurs at any processor i in view v (or i crashes), then a view change will eventually occur at all processors in $v.set - \{i\}$.

7.1.3 View-Graphs

We introduce *view-graphs* that represent view changes at processors in executions and that are used to analyze properties of executions. View-graphs are directed graphs (digraphs) that are defined by the states and by the NEWVIEW events of executions of algorithms that use group communication services. Representing view changes as digraphs enables us to use common graph analysis techniques to formally reason about the properties of executions. Our view-graph approach to the analysis of executions is general, and we believe it can be used to study other properties of group communication services and algorithms for partitionable networks.

Consider an algorithm Λ that uses a group communication service (GCS). We modify algorithm Λ by introducing, for each processor i, the history variable cv_i that keeps track of the current view at i as follows: In the initial state, we set cv_i to be v_0, the distinguished initial view for all processors $i \in \mathcal{P}$. In the effects of the $\text{NEWVIEW}(v)_i$ action for processor i, we include the assignment $cv_i := v$. From this point on (and until the end of Section 7.1) we assume that algorithms are modified to include such history variables. We now define *view-graphs* by specifying how a view-graph is induced by an execution of an algorithm.

Definition 7.1 Given an execution ξ of algorithm Λ, the *view-graph* $\Gamma_\xi = \langle V, E, L\rangle$ is defined to be the labeled directed graph as follows:

1. Let V_ξ be the set of all views v that occur in $\text{NEWVIEW}(v)_i$ events in ξ. The set V of nodes of Γ_ξ is the set $V_\xi \cup \{v_0\}$. We call v_0 the initial node of Γ_ξ.
2. The set of edges E of Γ_ξ is a subset of $V \times V$ determined as follows. For each $\text{NEWVIEW}(v)_i$ event in ξ that occurs in state s, the edge $(s.cv_i, v)$ is in E.
3. The edges in E are labeled by $L : E \to 2^{\mathcal{P}}$, such that $L(u, v) = \{i : \text{NEWVIEW}(v)_i$ occurs in state s in ξ such that $s.cv_i = u\}$.

Observe that the definition ensures that all edges are labeled.

Example 7.1 Consider the following execution ξ (we omit all events other than NEWVIEW and any states that do not precede NEWVIEW events).

$$\begin{aligned}\xi = s_0, &\text{NEWVIEW}(v_1)_{p_1}, \ldots, s_1, \text{NEWVIEW}(v_2)_{p_2}, \ldots, s_2, \text{NEWVIEW}(v_3)_{p_4}, \ldots,\\ &s_3, \text{NEWVIEW}(v_4)_{p_1}, \ldots, s_4, \text{NEWVIEW}(v_1)_{p_3}, \ldots, s_5, \text{NEWVIEW}(v_4)_{p_2}, \ldots,\\ &s_6, \text{NEWVIEW}(v_4)_{p_3}, \ldots\end{aligned}$$

Let $v_1.set = \{p_1, p_3\}$, $v_2.set = \{p_2\}$, $v_3.set = \{p_4\}$ and $v_4.set = \{p_1, p_2, p_3\}$. Additionally, $v_0.set = \mathcal{P} = \{p_1, p_2, p_3, p_4\}$.

The view-graph $\Gamma_\xi = \langle V, E, L\rangle$ is given in Figure 5. The initial node of Γ_ξ is v_0. The set of nodes of V of Γ_ξ is $V = V_\xi \cup \{v_0\} = \{v_0, v_1, v_2, v_3, v_4\}$. The set of edges E of Γ_ξ is $E = \{(v_0, v_1), (v_0, v_2), (v_0, v_3), (v_1, v_4), (v_2, v_4)\}$, since for each of these (v_k, v_ℓ) the event $\text{NEWVIEW}(v_\ell)_i$ occurs in state s_t where $s_t.cv_i = v_k$ for some certain i (by the definition of the history variable). The labels of the edges are $L(v_0, v_1) = \{p_1, p_3\}$, $L(v_0, v_2) = \{p_2\}$, $L(v_0, v_3) = \{p_4\}$, $L(v_1, v_4) = \{p_1, p_3\}$ and $L(v_2, v_4) = \{p_2\}$, since for each $p_i \in L(v_k, v_\ell)$ the event $\text{NEWVIEW}(v_\ell)_{p_i}$ occurs in state s_t where $s_t.cv_{p_i} = v_k$.

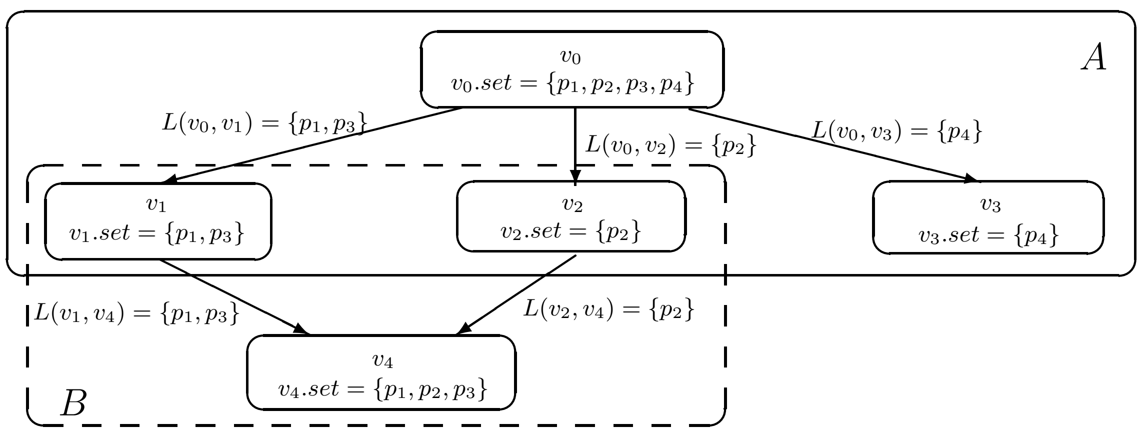

Figure 5: Example of a view-graph

We now show certain properties of view-graphs. Given a graph H and a node v of H, we define *indegree*(v, H) (*outdegree*(v, H)) to be the indegree (outdegree) of v in H.

Lemma 7.1 For any execution ξ, *indegree*$(v_0, \Gamma_\xi) = 0$.

Proof: In the initial state s_0, $s_0.cv$ is defined to be v_0 for all processors in $\mathcal{P}$ and $v_0.set = \mathcal{P}$. Assume that *indegree*$(v_0, \Gamma_\xi) > 0$. By the construction of view-graphs, this implies that some processor $i \in \mathcal{P}$ installs v_0 a second time. But this contradicts the property 2(a) of GCS. $\square$

Lemma 7.2 Let ξ be an execution and $\Gamma_\xi|_i$ be the projection of Γ_ξ on the edges whose label includes i, for some $i \in \mathcal{P}$. $\Gamma_\xi|_i$ is an elementary path and v_0 is the path's source node.

Proof: Let execution ξ be $s_0, e_1, s_1, e_2, \ldots$. Let $\xi^{(k)}$ be the prefix of ξ up to the k^{th} state. i.e., $\xi^{(k)} = s_0, e_1, s_1, e_2, \ldots, s_k$. Let Γ_ξ^k be the view-graph that is induced by $\xi^{(k)}$. Then define $\Gamma_\xi^k|_i$ to be the projection of Γ_ξ^k on the edges whose label includes i, for some $i \in \mathcal{P}$. For an elementary path π, we define $\pi.sink$ to be its sink node.

We prove by induction on k that $\Gamma_\xi^k|_i$ is an elementary path, that $\Gamma_\xi^k|_i.sink = s_k.cv_i$ and that v_0 is the path's source node.

Basis: $k = 0$. $\Gamma_\xi^0|_i$ has only one vertex, v_0, and no edges ($\xi^{(0)} = s_0$). Thus, $\Gamma_\xi^0|_i.sink = s_0.cv_i = v_0$ and v_0 is the source node of this path.

Inductive Hypothesis: Assume that $\forall n \leq k$, $\Gamma_\xi^n|_i$ is an elementary path, that $\Gamma_\xi^n|_i.sink$ $= s_n.cv_i$ and that v_0 is the path's source node.

Inductive Step: $n = k + 1$. For state s_{k+1} we consider two cases:

Case 1: If event e_{k+1} is not a NEWVIEW event involving processor i, then $\Gamma_\xi^{k+1}|_i = \Gamma_\xi^k|_i$. Thus, by inductive hypothesis, $\Gamma_\xi^{k+1}|_i$ is an elementary path and v_0 is its source node. From state s_k to state s_{k+1}, processor i did not witness any new view. By the definition of the history variable, $s_{k+1}.cv_i = s_k.cv_i$. Thus, $\Gamma_\xi^{k+1}|_i.sink = s_k.cv_i = s_{k+1}.cv_i$.

Case 2: If event e_{k+1} is a NEWVIEW$(v)_i$ event that involves processor i, then by the construction of the view-graph, $(s_k.cv_i, v)$ is a new edge from node $s_k.cv_i$ to node v. By inductive hypothesis, $\Gamma_\xi^k|_i.sink = s_k.cv_i$. Since our GCS does not allow the same view to be installed twice (property 2(a)), $v \neq u$ for all $u \in \Gamma_\xi^k|_i$. Thus, $\Gamma_\xi^{k+1}|_i$ is also an elementary path, with v_0 its source node and $\Gamma_\xi^{k+1}|_i.sink = v$. From state s_k to state s_{k+1}, processor i installs the new view v. By the definition of the history variable, $s_{k+1}.cv_i = v$. Thus, $\Gamma_\xi^{k+1}|_i.sink = s_{k+1}.cv_i$. This completes the proof. □

Theorem 7.3 Any view-graph Γ_ξ, induced by any execution ξ of algorithm Λ is a connected graph.

Proof: The result follows from Definition 7.1(2), from the observation that all edges of the view-graph are labeled and from Lemma 7.2 □

We now demonstrate how we can use view-graphs to represent group fragmentations and merges. We begin with fragmentations.

Definition 7.2 For a view-graph $\Gamma_\xi = \langle V, E, L\rangle$, a *fragmentation subgraph* is a connected labeled subgraph $H = \langle V_H, E_H, L_H\rangle$ of Γ_ξ such that:

1. H contains a unique node v such that $indegree(v, H) = 0$; v is called the *fragmentation node* of H.
2. $V_H = \{v\} \cup V'_H$, where V'_H is defined to be $\{w : (v, w) \in E\}$.
3. $E_H = \{(v, w) : w \in V'_H\}$.
4. L_H is the restriction of L on E_H.
5. $\bigcup_{w\in V'_H}(w.set) = v.set$.
6. $\forall u, w \in V'_H$ such that $u \neq w,\ u.set \cap w.set = \emptyset$.
7. $\forall w \in V'_H$, $L_H(v, w) = w.set$.

We refer to all NEWVIEW events that collectively induce a fragmentation subgraph for a fragmentation node v as a *fragmentation*.

Example 7.2 Area A in Figure 5 (solid box) shows the fragmentation subgraph $H = \langle V_H, E_H, L_H \rangle$ of Γ_ξ from Example 7.1. Here $V_H = \{v_0, v_1, v_2, v_3\}$, $E_H = \{(v_0, v_1), (v_0, v_2), (v_0, v_3)\}$ and the labels are the labels of Γ_ξ restricted on E_H. We can confirm that H is a fragmentation subgraph by examining the individual items of Definition 7.2.

We continue with the representation of group merges using view-graphs.

Definition 7.3 For a view-graph $\Gamma_\xi = \langle V, E, L \rangle$, a *merge subgraph* is a connected labeled subgraph $H = \langle V_H, E_H, L_H \rangle$ of Γ_ξ such that:

1. H contains a unique node v such that *outdegree*$(v, H) = 0$ and *indegree*$(v, H) > 1$; v is called the *merge node* of H.
2. $V_H = \{v\} \cup V'_H$, where V'_H is defined to be $\{w : (w, v) \in E\}$.
3. $E_H = \{(w, v) : w \in V'_H\}$.
4. L_H is the restriction of L on E_H.
5. $\bigcup_{w \in V'_H} (w.set) = v.set$.
6. $\forall u, w \in V'_H$ such that $u \neq w,\ u.set \cap w.set = \emptyset$.
7. $\bigcup_{w \in V'_H} L_H(w, v) = v.set$.

We refer to all NEWVIEW events that collectively induce a merge subgraph for a merge node v as a *merge*.

Note that a regrouping of a group g_1 to a group g_2 such that $g_1.set = g_2.set$ can be represented either as a fragmentation subgraph (fragmentation) or as a merge subgraph (merge). Following the convention established in the definition of adversary $\mathcal{A}_{FM}$ (Section 3.2.2), we represent it as a fragmentation subgraph by requiring that *indegree*$(v, H) > 1$ for any merge node v.

Example 7.3 Area B in Figure 5 (dashed box) of Example 7.1 shows the merge subgraph $H = \langle V_H, E_H, L_H \rangle$ of Γ_ξ, where $V_H = \{v_1, v_2, v_3, v_4\}$, $E_H = \{(v_1, v_4), (v_2, v_4)\}$ and the labels are the labels of Γ_ξ restricted on E_H. We can verify this by examining all conditions of Definition 7.3.

We now give some additional definitions and show that any view graph is a directed acyclic graph (DAG).

Definition 7.4 Given a view-graph Γ_ξ we define:

(a) *frag*(Γ_ξ) to be the set of all the distinct fragmentation nodes in Γ_ξ,

(b) *merg*(Γ_ξ) to be the set of all the distinct merge nodes in Γ_ξ.

Definition 7.5 Given a view-graph Γ_ξ:

(a) if all of its non-terminal nodes are in *frag*(Γ_ξ), then Γ_ξ is called a *fragmentation view-graph*.

(b) if each of its non-terminal nodes is either in *frag*(Γ_ξ), or it is an immediate ancestor of a node which is in *merg*(Γ_ξ), then Γ_ξ is called an *fm view-graph*.

For Γ_ξ in the example in Figure 5 we have $v_0 \in$ *frag*(Γ_ξ) by Definition 7.4(a). Also, $v_4 \in merg(\Gamma_\xi)$ per Definition 7.4(b); additionally, the nodes v_1 and v_2 are immediate ancestors of $v_4 \in merg(\Gamma_\xi)$. By Definition 7.5(b), Γ_ξ is an fm view-graph. Observe that Γ_ξ is a DAG. This is true for all view-graphs:

Theorem 7.4 Any view-graph $\Gamma_\xi = \langle V, E, L \rangle$ is a Directed Acyclic Graph (DAG).

Proof: Assume that Γ_ξ is not a DAG. Thus, it contains at least one cycle. Let $((v_1, v_2)(v_2, v_3) \ldots (v_k, v_1))$ be an elementary cycle of Γ_ξ. By the construction of view-graphs

(Definition 7.1(3)) and by the monotonicity property (property 2) of GCS, $v_i.id < v_{i+1}.id$ for $1 \leq i \leq k$ and $v_k.id < v_1.id$. But, by the transitivity of "$<$", $v_1.id < v_k.id$, a contradiction. $\Box$

Corollary 7.5 Any fm view graph is a DAG and any fragmentation view-graph is a rooted tree.

In the complexity analysis of our algorithm, we exploit the fact that view graphs are DAGs. In particular we use the following fact.

Fact 7.1 In any (non-empty) DAG, there is at least one vertex, such that all of its descendants have outdegree 0.

Remark 7.1 Consider an execution ξ of algorithm Λ under adversary $\mathcal{A}_{FM}$. In Section 3.2.2 we defined the fragmentation-number $f_r(\xi|_{\mathcal{A}_{FM}})$ and merge-number $f_m(\xi|_{\mathcal{A}_{FM}})$ of the adversarial pattern $\xi|_{\mathcal{A}_{FM}}$ of execution ξ. We can also use view-graphs to define these quantities. Namely, $f_r(\xi|_{\mathcal{A}_{FM}}) = |\{w : \text{NEWVIEW}(w)_i$ occurs in $\xi \wedge (v,w) \in E \wedge v \in \mathit{frag}(\Gamma_\xi)\}|$, and $f_m(\xi|_{\mathcal{A}_{FM}}) = |\{v : \text{NEWVIEW}(v)_i$ occurs in $\xi \wedge v \in \mathit{merg}(\Gamma_\xi)\}|$, where Γ_ξ is the view-graph of execution ξ.

7.1.4 Algorithm AX

We present Algorithm AX, that deals with fragmentations and merges and that relies on the GCS as specified in Section 7.1.2, and prove its correctness. We give its complexity analysis in Section 7.1.5.

7.1.4.1 Description of the Algorithm

Algorithm AX uses a coordinator approach within each group view. The high level idea of the algorithm is that each processor performs (remaining) tasks according to a load balancing rule, and a processor completes its computation when it learns the results of all the tasks.

Task allocation. The set T of the initial tasks is known to all processors. During the execution each processor i maintains a local set D of tasks already done, a local set R of the corresponding results, and the set G of processors in the current group. (The set D may be an underestimate of the set of tasks done globally.) The processors allocate tasks based on the shared knowledge of the processors in G about the tasks done. For a processor i, let $rank(i, G)$ be the rank of i in G when processor identifiers are sorted in ascending order. Let U be the tasks in $T - D$. For a task u in U, let $rank(u, U)$ be the rank of u in U when task identifiers are sorted in ascending order. Our *load balancing rule* for each processor i in G is that:

- if $rank(i, G) \leq |U|$, then processor i performs task u such that $rank(u, U) = rank(i, G)$;
- if $rank(i, G) > |U|$, then processor i does nothing.

Algorithm structure. The algorithm code is given in Figure 6 using Input/Output automata notation [81]. The algorithm uses the group communication service to structure its computation in terms of *rounds* numbered sequentially within each group view.

Initially all processors are members of the distinguished initial view v_0, such that $v_0.set = \mathcal{P}$. Rounds numbered 1 correspond to the initial round either in the original group or in a new group upon a regrouping as notified via the NEWVIEW event. If a regrouping occurs, the processor receives the new set of members from the group membership service and starts the

Data types and identifiers:

$\mathcal{T}$: tasks
$\mathcal{R}$: results
$Result : \mathcal{T} \to \mathcal{R}$
$\mathcal{M}es$: messages
$\mathcal{P}$: processor ids
$\mathcal{G}$: group ids
$views = \mathcal{G} \times 2^{\mathcal{P}}$: views, selectors id and set

$m \in \mathcal{M}es$
$i, j \in \mathcal{P}$
$v \in views$
$Z \in 2^{\mathcal{T}}$
$Q \in 2^{\mathcal{R}}$
$round \in \mathbb{N}$
$results \in 2^{\mathcal{R}}$

States:

$T \in 2^{\mathcal{T}}$, the set of $n = |T|$ tasks
$D \in 2^{\mathcal{T}}$, the set of done tasks, initially $\emptyset$
$R \in 2^{\mathcal{R}}$, the set of known results, initially $\emptyset$
$G \in 2^{\mathcal{P}}$, current members, init. $v_0.set = \mathcal{P}$
$X \in 2^{\mathcal{M}es}$, messages since last NEWVIEW, initially $\emptyset$
$Rnd \in \mathbb{N}$, round number, initially 1
$Phase \in \{send, receive, sleep, mcast, mrecv\}$, initially $send$

Derived variables:

$U = T - D$, the set of remaining tasks
$Coordinator(i)$: Boolean,
  if $i = \max_{j \in G}\{j\}$
  then *true* else *false*
$Next(U, G)$, next task u, such that
  $rank(u, U) = rank(i, G)$

History variables:

$cv_i \in views$ $(i \in \mathcal{P})$,
  initially $\forall i,\ cv_i = v_0$.
$\text{MSG}_i \in 2^{\mathcal{M}es}$ $(i \in \mathcal{P})$,
  initially $\forall i,\ \text{MSG}_i = \emptyset$.

Transitions at i:

input NEWVIEW$(v)_i$
Effect:
  $G \leftarrow v.set$
  $X \leftarrow \emptyset$
  $Rnd \leftarrow 1$
  $Phase \leftarrow send$
  $cv := v$

output GP1SND$(m, j)_i$
Precondition:
  $Coordinator(j)$
  $Phase = send$
  $m = \langle i, D, R, Rnd \rangle$
Effect:
  $\text{MSG} := \text{MSG} \cup \{m\}$
  $Phase \leftarrow receive$

input GP1RCV$(\langle j, Z, Q, round \rangle)_i$
Effect:
  $X \leftarrow X \cup \{\langle j, Z, Q, round \rangle\}$
  $R \leftarrow R \cup Q$
  $D \leftarrow D \cup Z$
  if $G = \{j : \langle j, *, *, Rnd \rangle \in X\}$ then
    $Phase \leftarrow mcast$

output GPMSND$(m)_i$
Precondition:
  $Coordinator(i)$
  $m = \langle i, D, R, Rnd \rangle$
  $Phase = mcast$
Effect:
  $\text{MSG} := \text{MSG} \cup \{m\}$
  $Phase \leftarrow mrecv$

input GPMRCV$(\langle j, Z, Q, round \rangle)_i$
Effect:
  $D \leftarrow D \cup Z$
  $R \leftarrow R \cup Q$
  if $D = T$ then
    $Phase \leftarrow sleep$
  else
    if $rank(i, G) < |U|$ then
      $R \leftarrow R \cup \{Result(Next(U, G))\}$
      $D \leftarrow D \cup \{Next(U, G)\}$
    $Rnd \leftarrow Rnd + 1$
    $Phase \leftarrow send$

Figure 6: Algorithm AX.

first round of this view (NEWVIEW action). At the beginning of each round, denoted by a round number Rnd, processor i knows G, the local set D of tasks already done, and the set R of the results. Since all processors know G, they "elect" the group coordinator to be the processor which has the highest processor id (no communication is required since the coordinator is uniquely identified). In each round each processor reports D and R to the coordinator of G (GP1SND action). The coordinator receives and collates these reports (GP1RCV action) and sends the result to the group members (GPMSND action). Upon the receipt of the message from the coordinator, processors update their D and R, and perform work according to the load balancing rule (GPMRCV action).

For generality, we assume that the messages may be delivered by the GCS out of order. The set of messages within the current view is saved in the local variable X. The saved messages are also used to determine when all messages for a given round have been received. Processing continues until each member of G knows all results (the processors enter the *sleep* stage).

The variables cv and MSG are *history variables* that do not affect the algorithm, but play a role in its analysis.

7.1.4.2 Correctness of the Algorithm

We now show the safety of algorithm AX. We first show that no processor stops working as long as it knows of any undone tasks.

Theorem 7.6 (Safety 1) For all states of any execution of Algorithm AX it holds that

$$\forall i \in \mathcal{P} : D_i \neq T \Rightarrow Phase \neq sleep.$$

Proof: The proof follows by examination of the code of the algorithm, and more specifically from the code of the input action GPMRCV$(\langle j, Z, Q, round\rangle)_i$. $\square$

Note that the implication in Theorem 7.6 cannot be replaced by iff ($\Leftrightarrow$). This is because if $D_i = T$, we may still have $Phase \neq sleep$. This is the case where processor i becomes a member of a group in which the processors do not know all the results of all the tasks.

Next we show that if some processor does not know the result of some task, this is because it does not know that this task has been performed (Theorem 7.8 below). We show this using the history variables MSG_i $(i \in \mathcal{P})$.

We define MSG_i to be a history variable that keeps on track all the messages sent by processor $i \in \mathcal{P}$ in all GP1SND and GPMSND events of an execution of algorithm AX. Formally, in the effects of the $\text{GP1SND}(m, j)_i$ and $\text{GPMSND}(m)_i$ actions we include the assignment $\text{MSG}_i := \text{MSG}_i \cup \{m\}$. Initially, $\text{MSG}_i = \emptyset$ for all i. We define $\mathcal{MSG}$ to be $\bigcup_{i \in \mathcal{P}} \text{MSG}_i$.

Lemma 7.7 If m is a message received by processor $i \in \mathcal{P}$ in a $\text{GP1RCV}(m)_i$ or $\text{GPMRCV}(m)_i$ event of an execution of algorithm AX, then $m \in \mathcal{MSG}$.

Proof: Property 3 of the GCS (Section 7.1.2) requires that for every receive event there exists a preceding send event of the same message (the GCS does not generate messages). Hence, m must have been sent by some processor $j \in \mathcal{P}$ (possible $j = i$) in some earlier event of the execution. Messages can be sent only in $\text{GP1SND}(m, i)_j$ or $\text{GPMSND}(m)_j$ events. By definition, $m \in \text{MSG}_j$. Hence, $m \in \mathcal{MSG}$. $\square$

Theorem 7.8 (Safety 2) For all states of any execution of Algorithm AX:

(a) $\forall t \in T,\ \forall i \in \mathcal{P} : result(t) \notin R_i \Rightarrow t \notin D_i$, and

(b) $\forall t \in T, \forall \langle i, D', R', Rnd \rangle \in \mathcal{MSG} : result(t) \notin R' \Rightarrow t \notin D'$.

Proof: Let ξ be an execution of AX and ξ^k be the prefix of ξ up to the k^{th} state, i.e., $\xi^k = s_0, e_1, s_1, e_2, \ldots, s_k$. The proof is done by induction on k.

Base Case: $k = 0$. In s_0, $\forall i \in \mathcal{P}, D_i = \emptyset, R_i = \emptyset$ and $\mathcal{MSG} = \emptyset$.

Inductive Hypothesis: For a state s_ℓ such that $\ell \leq k$, $\forall t \in T$, $\forall i \in \mathcal{P} : result(t) \notin R_i \Rightarrow t \notin D_i$, and $\forall t \in T, \forall \langle i, D', R', Rnd \rangle \in \mathcal{MSG} : result(t) \notin R' \Rightarrow t \notin D'$.

Inductive Step: $\ell = k + 1$. Consider the following seven types of actions leading to the state s_{k+1}:

1. $e_{k+1} = \text{NEWVIEW}(v')_i$: The effect of this action does not affect the invariant. By the inductive hypothesis, in state s_{k+1}, the invariant holds.
2. $e_{k+1} = \text{GP1SND}(m, j)_i$: Clearly, the effect of this action does not affect part (a) of the invariant but it affects part (b). Since $m = \langle i, D_i, R_i, Rnd \rangle$, by the inductive hypothesis part (a), the assignment $m \in \mathcal{MSG}$ reestablishes part (b) of the invariant. Thus, in state s_{k+1}, the invariant is reestablished.
3. $e_{k+1} = \text{GP1RCV}(\langle j, Z, Q, round \rangle)_i$: Processor i updates R_i and D_i according to Q and Z respectively. The action is atomic, i.e., if R_i is updated, then D_i must be also updated. By Lemma 7.7, $\langle j, Z, Q, round \rangle \in \mathcal{MSG}$. Thus, by the inductive hypothesis part (b), $\forall t \in T : result(t) \notin Z \Rightarrow t \notin Q$. From the fact that D_i and R_i are updated according to Z and Q respectively and by the inductive hypothesis part (a), in state s_{k+1}, the invariant is reestablished.
4. $e_{k+1} = \text{GPMSND}(m)_i$: Clearly, the effect of this action does not affect part (a) of the invariant but it affects part (b). Since $m = \langle i, D_i, R_i, Rnd \rangle$, by the inductive hypothesis part (a), the assignment $m \in \mathcal{MSG}$ reestablishes part (b) of the invariant. Thus, in state s_{k+1}, the invariant is reestablished.
5. $e_{k+1} = \text{GPMRCV}(\langle j, Z, Q, round \rangle)_i$: By Lemma 7.7, $\langle j, Z, Q, round \rangle \in \mathcal{MSG}$. By the inductive hypothesis part (b), $\forall t \in T : result(t) \notin Z \Rightarrow t \notin Q$. Processor i updates R_i and D_i according to Q and Z respectively. Since Z and Q have the required property, by

the inductive hypothesis part (a), the assignments to D_i and R_i reestablish the invariant. In the case where $D_i \neq T$, processor i performs a task according to the load balancing rule. Let $u \in T$ be this task. Because of the action atomicity, when processor i updates R_i with $result(u)$, it must also update D_i with u. Hence, in state s_{k+1}, the invariant is reestablished.

6. $e_{k+1} = \text{REQUEST}_{q,i}$: The effect of this action does not affect the invariant.
7. $e_{k+1} = \text{REPORT}(results)_{q,i}$: The effect of this action does not affect the invariant.

This completes the proof. □

7.1.5 Analysis of Algorithm AX

Per Definition 3.6, we express the task-oriented work complexity of algorithm AX under adversary $\mathcal{A}_{FM}$ as $W_{\mathcal{A}_{FM}}(n, p, f) = W_{\mathcal{A}_{FM}}(n, p, f_r + f_m)$, where f_r and f_m is the fragmentation-number and merge-number, respectively, of the execution of algorithm AX that maximizes work. Per Definition 3.7, the message complexity is expressed as $M_{\mathcal{A}_{FM}}(n, p, f) = M_{\mathcal{A}_{FM}}(n, p, f_r + f_m)$. Our analysis focuses on assessing the impact of the fragmentation-number f_r and the merge-number f_m on the work and message complexity, and in the rest of this section for clarity we let $\mathcal{W}_{f_r,f_m}$ stand for $W_{\mathcal{A}_{FM}}(n, p, f_r + f_m)$, and $\mathcal{M}_{f_r,f_m}$ stand for $M_{\mathcal{A}_{FM}}(n, p, f_r + f_m)$.

7.1.5.1 Work Complexity

We begin the analysis of algorithm AX by first providing definitions and then proving several lemmas that lead to the work complexity of the algorithm.

Definition 7.6 Let ξ^μ be any execution of algorithm AX in which all the processors learn the results of all tasks and that includes a merge of groups $g_1, \ldots, g_k$ into the group μ, where the processors in μ undergo no further view changes. We define $\bar{\xi}^\mu$ to be the execution we derive by removing the merge from ξ^μ as follows: (1) We remove all states and events that correspond to the merge of groups $g_1, \ldots, g_k$ into the group μ and all states and events for processors within μ. (2) We add the appropriate states and events such that the processors in groups $g_1, \ldots, g_k$ undergo no further view changes and perform any remaining tasks.

Definition 7.7 Let ξ^φ be any execution of algorithm AX in which all the processors learn the results of all tasks and that includes a fragmentation of the group φ to the groups $g_1, \ldots, g_k$ where the processors in these groups undergo no further view changes. We define $\bar{\xi}^\varphi$ to be the execution we derive by removing the fragmentation from ξ^φ as follows: (1) We remove all states and events that correspond to the fragmentation of the group φ to the groups $g_1, \ldots, g_k$ and all states and events of the processors within the groups $g_1, \ldots, g_k$. (2) We add the appropriate states and events such that the processors in the group φ undergo no further view changes and perform any remaining tasks.

Note: In Definitions 7.6 and 7.7, we claim that we can remove states and events from an execution and add some other states and events to it. This is possible because if the processors in a single view installed that view and there are no further view changes, then the algorithm will continue making computation progress. So, if we remove all states and events corresponding to a view change, then the algorithm can always proceed as if this view change never occurred.

Lemma 7.9 In algorithm AX, for any view v, including the initial view, if the group is not subject to any regroupings, then the work required to complete all tasks in the view is no more than $n - \max_{i \in v.set}\{|D_i|\}$, where D_i is the value of the state variable D of processor i at the start of its local round 1 in view v.

Proof: In the first round, all the processors send messages to the coordinator containing D_i. The coordinator computes $\cup_{i \in v.set}\{D_i\}$ and broadcasts this result to the group members. Since the group is not subject to any regroupings, the number of tasks t, that the processors need to perform is: $t = n - |\cup_{i \in v.set}\{D_i\}|$. In each round of the computation, by the load balancing rule, the members of the group perform distinct tasks and no task is performed more than once. Therefore, t is the work performed in this group. On the other hand, $\max_{i \in v.set}\{|D_i|\} \leq |\cup_{i \in v.set}\{D_i\}|$, thus, $t \leq n - \max_{i \in v.set}\{|D_i|\}$. $\square$

In the following lemma, groups $\mu, g_1, \ldots, g_k$ are defined as in Definition 7.6.

Lemma 7.10 Let ξ^μ be an execution of Algorithm AX as in Definition 7.6. Let W_1 be the work performed by the algorithm in the execution ξ^μ. Let W_2 be the work performed by Algorithm AX in the execution $\bar{\xi}^\mu$. Then $W_1 \leq W_2$.

Proof: For the execution ξ^μ, let W' be the work performed by the processors in $\mathcal{P} - \bigcup_{1 \leq i \leq k}(g_i.set) - \mu.set$. Observe that the work performed by the processors in $\mathcal{P} - \bigcup_{1 \leq i \leq k}(g_i.set)$ in the execution $\bar{\xi}^\mu$ is equal to W'. The work that is performed by processor j in $g_i.set$ prior to the $\text{NEWVIEW}(\mu)_j$ event in ξ^μ, is the same in both executions. Call this work $W_{i,j}$. Define $W'' = \sum_{i=1}^{k} \sum_{j \in g_i.set} W_{i,j}$. Define $W_s = W' + W''$. Thus, W_s is the same in both executions, ξ^μ and $\bar{\xi}^\mu$. Define W_μ to be the work performed by all processors in $\mu.set$ in execution ξ^μ. For each processor j in $g_i.set$, let D_j be the value of the state variable D just prior to the $\text{NEWVIEW}(\mu)_j$ event in ξ^μ. For each g_i, define: $d_i = |\bigcup_{j \in g_i.set} D_j|$. Thus there are at least $n - d_i$ tasks that remain to be done in each g_i.

In execution $\bar{\xi}^{\mu}$, the processors in each group g_i proceed and complete these remaining tasks. This requires work at least $n - d_i$. Define this work as W_{g_i}. Thus, $W_{g_i} \geq (n - d_i)$. In execution ξ^{μ}, groups $g_1, \ldots, g_k$ merge into group μ. The number of tasks that need to be performed by the members of μ is at most $n - d_j$, where $d_j = \max_i\{d_i\}$ for some j. By Lemma 7.9, $W_{\mu} \leq n - d_j$. Observe that:

$$W_1 = W_s + W_{\mu} \leq W_s + n - d_j \leq W_s + \sum_{i=1}^{k}(n - d_i) \leq W_s + \sum_{i=1}^{k} W_{g_i} = W_2,$$

as desired. □

In the following lemma, groups $\varphi, g_1, \ldots, g_k$ are defined as in Definition 7.7.

Lemma 7.11 Let ξ^{φ} be an execution of Algorithm AX as in Definition 7.7. Let W_1 be the work performed by the algorithm in the execution ξ^{φ}. Let W_2 be the worked performed by Algorithm AX in the execution $\bar{\xi}^{\varphi}$. Then $W_1 \leq W_2 + W_3$, where W_3 is the work performed by all processors in $\bigcup_{1 \leq i \leq k}(g_i.set)$ in the execution ξ^{φ}.

Proof: Let W' be the work performed by all processors in $\mathcal{P} - \bigcup_{1 \leq i \leq k}(g_i.set) - \varphi.set$ in the execution ξ^{φ}. Observe that the work performed by all processors in $\mathcal{P} - \varphi.set$ in the execution $\bar{\xi}^{\varphi}$ is equal to W'. The work that is performed by processor j in $\varphi.set$ prior to the $\text{NEWVIEW}(g_i)_j$ event in ξ^{φ}, is the same in both executions. Call this work $W_{\varphi,j}$. Define $W'' = \sum_{j \in \varphi.set} W_{\varphi,j}$. Define $W_s = W' + W''$. Thus, W_s is the same in both executions, ξ^{φ} and $\bar{\xi}^{\varphi}$. Define W_{φ} to be the work performed by all processors in $\varphi.set$ in execution $\bar{\xi}^{\varphi}$. Let $W''' = W_{\varphi} - W''$. Observe that:

$$W_1 = W_s + W_3 \leq W_s + W_3 + W''' = W_2 + W_3,$$

as desired. □

Lemma 7.12 $\mathcal{W}_{f_r,f_m} \leq n \cdot p$.

Proof: By the construction of algorithm AX, when processors are not able to exchange information about task execution due to regroupings, in the worst case, each processor has to perform all n tasks by itself. Since we can have at most p processors doing that the result follows. □

Lemma 7.13 $\mathcal{W}_{f_r,f_m} \leq n \cdot f_r + n$.

Proof: By induction on the number of views, denoted by ℓ, occurring in an execution. For a specific execution ξ_ℓ with ℓ views, let $f_r(\xi_\ell) = f_r^{(\ell)}$ be the fragmentation-number and $f_m(\xi_\ell) = f_m^{(\ell)}$ the merge-number.

Base Case: $\ell = 0$. Since $f_r^{(\ell)}$ and $f_m^{(\ell)}$ must also be 0, the base case follows from Lemma 7.9.

Inductive Hypothesis: Assume that for all $\ell \leq k$, $\mathcal{W}_{f_r^{(\ell)},f_m^{(\ell)}} \leq n \cdot f_r^{(\ell)} + n$.

Inductive Step: Need to show that for $\ell = k+1$, $\mathcal{W}_{f_r^{(k+1)},f_m^{(k+1)}} \leq n \cdot f_r^{(k+1)} + n$.

Consider a specific execution ξ_{k+1} with $\ell = k+1$. Let $\Gamma_{\xi_{k+1}}$ be the view-graph induced by this execution. The view-graph has at least one vertex such that all of its descendants are sinks (Fact 7.1). Let ν be such a vertex. We consider two cases:

Case 1: Vertex ν has a descendant μ that corresponds to a merge in the execution. Therefore all ancestors of μ in $\Gamma_{\xi_{k+1}}$ have outdegree 1. Since μ is a sink vertex, the group that corresponds to μ performs all the remaining (if any) tasks and does not perform any additional work. Let $\xi_k = \bar{\xi}^{\mu}_{k+1}$ (per Definition 7.6) be an execution in which this merge does not occur. In execution ξ_k, the number of views is k. Also, $f_r^{(k+1)} = f_r^{(k)}$ and $f_m^{(k+1)} = f_m^{(k)} + 1$. By inductive hypothesis, $\mathcal{W}_{f_r^{(k)},f_m^{(k)}} \leq n \cdot f_r^{(k)} + n$. By Lemma 7.10, the work performed in execution ξ_{k+1}, is no worse than the work performed in execution ξ_k. Hence, the total work complexity is:

$$\mathcal{W}_{f_r^{(k+1)},f_m^{(k+1)}} \leq \mathcal{W}_{f_r^{(k)},f_m^{(k)}} \leq n \cdot f_r^{(k)} + n = n \cdot f_r^{(k+1)} + n.$$

Case 2: Vertex ν has no descendants that correspond to a merge in the execution. Therefore, the group that corresponds to ν must fragment, say into q groups. These groups correspond to sink vertices in $\Gamma_{\xi_{k+1}}$, thus they perform all the remaining (if any) tasks and do not perform any additional work. Let $\xi_{k+1-q} = \bar{\xi}^{\nu}_{k+1}$ (per Definition 7.7) be an execution in which the fragmentation does not occur. In execution ξ_{k+1-q}, the number of views is $k+1-q \leq k$. Also, $f_r^{(k+1-q)} = f_r^{(k+1)} - q$ and $f_m^{(k+1-q)} = f_m^{(k+1)}$. By inductive hypothesis, $\mathcal{W}_{f_r^{(k+1-q)}, f_m^{(k+1-q)}} \leq n \cdot f_r^{(k+1-q)} + n$. From Lemma 7.9, the work performed in each new group caused by the fragmentation is no more than n. Let W_σ be the total work performed in all q groups. Thus, $W_\sigma \leq qn$. By Lemma 7.11, the work performed in execution ξ_{k+1}, is no worse than the work performed in execution ξ_{k+1-q} and the work performed in all q groups. Hence, the total work complexity is:

$$\begin{aligned}
\mathcal{W}_{f_r^{(k+1)}, f_m^{(k+1)}} &\leq \mathcal{W}_{f_r^{(k+1-q)}, f_m^{(k+1-q)}} + W_\sigma &&\leq n \cdot f_r^{(k+1-q)} + n + W_\sigma \\
&= n \cdot \left(f_r^{(k+1)} - q\right) + n + W_\sigma &&\leq n \cdot \left(f_r^{(k+1)} - q\right) + n + qn \\
&= n f_r^{(k+1)} - qn + n + qn &&= n \cdot f_r^{(k+1)} + n.
\end{aligned}$$

This completes the inductive proof. □

Note that it is not difficult to see that if $f \geq p$, then there exists an adversarial strategy that can cause any *Omni-Do* algorithm to have task-oriented work $\Omega(n \cdot p)$ (the adversary can arrange so that all processors work in isolation for the entire computation). Similarly, if $f < p$, then there exists an adversarial strategy that can cause any *Omni-Do* algorithm to have task-oriented work $\Omega(n \cdot f + n)$ (the adversary partitions the processors in f groups at the beginning of the computation, and then lets the f groups to run in isolation for the remainder of the computation). Therefore, $\Omega(\min\{n \cdot f + n, n \cdot p\})$ is a lower bound on the task-oriented work for *Omni-Do*. We now show that algorithm AX is *optimal* under adversary $\mathcal{A}_{FM}$.

Theorem 7.14 Algorithm AX solves the asynchronous *Omni-Do*$_{\mathcal{A}_{FM}}(n, p, f)$ problem with task-oriented work

$$\mathcal{W}_{f_r,f_m} \leq \min\{n \cdot f_r + n,\ n \cdot p\}.$$

Proof: It follows directly from Lemmas 7.12 and 7.13. □

Observe that $\mathcal{W}_{f_r,f_m}$ does not depend on f_m (this of course does not imply that for any given execution, the work does not depend on merges). This observation substantiates the intuition that merges lead to a more efficient computation.

7.1.5.2 Message Complexity

We start by showing several lemmas that lead to the message complexity of the algorithm.

Lemma 7.15 For algorithm AX, in any view v, including the initial view, if the group is not subject to any regroupings, and for each processor $i \in v.set$, D_i is the value of the state variable D at the start of its local round 1 in view v, then the number of messages M that are sent until all tasks are completed is $2(n - d) \leq M < 2(q + n - d)$ where $q = |v.set|$, and $d = |\bigcup_{i \in v.set} D_i|$.

Proof: By the load balancing rule, the algorithm needs $\lceil \frac{n-d}{q} \rceil$ rounds to complete all tasks. In each round each processor sends one message to the coordinator and the coordinator responds with a single message to each processor. Thus, $M = 2q \cdot (\lceil \frac{n-d}{q} \rceil)$. Using the properties of the *ceiling*, we get: $2(n - d) \leq M < 2(q + n - d)$. □

In the following lemma, groups $\mu, g_1, \ldots, g_k$ are defined as in Definition 7.6.

Lemma 7.16 Let ξ^μ be an execution of Algorithm AX as in Definition 7.6. Let M_1 be the message cost of the algorithm in the execution ξ^μ. Let M_2 be the message cost of Algorithm AX in the execution $\bar{\xi}^\mu$. Then $M_1 < M_2 + 2p$.

Proof: For the execution ξ^{μ}, let M' be the number of messages sent by the processors in $\mathcal{P} - \bigcup_{1 \le i \le k}(g_i.set) - \mu.set$. Observe that the number of messages sent by the processors in $\mathcal{P} - \bigcup_{1 \le i \le k}(g_i.set)$ in the execution $\bar{\xi}^{\mu}$ is equal to M'.

The number of messages sent by any processor j in $g_i.set$ prior to the $\text{NEWVIEW}(\mu)_j$ event in ξ^{μ}, is the same in both executions. Call this message cost $M_{i,j}$. Define $M'' = \sum_{i=1}^{k} \sum_{j \in g_i.set} M_{i,j}$. Define $M_s = M' + M''$. Thus, M_s is the same in both executions, ξ^{μ} and $\bar{\xi}^{\mu}$. Define M_{μ} to be the number of messages sent by all processors in $\mu.set$ in execution ξ^{μ}. For each processor j in $g_i.set$, let D_j be the value of the state variable D just prior to the $\text{NEWVIEW}(\mu)_j$ event in ξ^{μ}. For each g_i, define $d_i = |\bigcup_{j \in g_i.set} D_j|$. Thus there are at least $n - d_i$ tasks that remain to be done in each g_i.

In execution $\bar{\xi}^{\mu}$, the processors in each group g_i proceed and complete these remaining tasks. Let M_{g_i} be the number of messages sent by all processors in $g_i.set$ in order to complete the remaining tasks. By Lemma 7.15, $M_{g_i} \ge 2(n - d_i)$. In execution ξ^{μ}, groups $g_1, \ldots, g_k$ merge into group μ. The number of tasks that need to be performed by the members of μ is at most $n - d_j$, where $d_j = \max_i\{d_i\}$ for some j. By Lemma 7.15, $M_{\mu} < 2(q + n - d_j)$, where $q = |\mu.set|$. Observe that:

$$\begin{aligned} M_1 &= M_s + M_{\mu} & &< M_s + 2(q + n - d_j) \\ &\le M_s + 2q + 2\textstyle\sum_{i=1}^{k}(n - d_i) & &\le M_s + 2q + \textstyle\sum_{i=1}^{k} M_{g_i} \\ &= M_2 + 2q & &\le M_2 + 2p, \end{aligned}$$

as desired. $\square$

In the following lemma, groups $\varphi, g_1, \ldots, g_k$ are defined as in Definition 7.7.

Lemma 7.17 Let ξ^{φ} be an execution of Algorithm AX as in Definition 7.7. Let M_1 be the message cost of the algorithm in the execution ξ^{φ}. Let M_2 be the message cost of Algorithm

AX in the execution $\bar{\xi}^{\varphi}$. Then $M_1 \leq M_2 + M_3$, where M_3 is the number of messages sent by all processors in $\bigcup_{1\leq i\leq k}(g_i.set)$ in the execution ξ^{φ}.

Proof: For the execution ξ^{φ}, let M' be the number of messages sent by the processors in $\mathcal{P} - \bigcup_{1\leq i\leq k}(g_i.set) - \varphi.set$. Observe that the number of messages sent by the processors in $\mathcal{P} - \varphi.set$ in the execution $\bar{\xi}^{\varphi}$ is equal to M'. The number of messages sent by processor j in $\varphi.set$ prior to the $\text{NEWVIEW}(g_i)_j$ event in ξ^{φ}, is the same in both executions. Call this message cost $M_{\varphi,j}$. Define $M'' = \sum_{j\in\varphi.set} M_{\varphi,j}$. Define $M_s = M' + M''$. Thus, M_s is the same in both executions, ξ^{φ} and $\bar{\xi}^{\varphi}$. Define M_{φ} to be the number of messages sent by all processors in $\varphi.set$ in execution $\bar{\xi}^{\varphi}$. Let $M''' = M_{\varphi} - M''$. Observe that:

$$M_1 = M_s + M_3 \leq M_s + M_3 + M''' = M_2 + M_3,$$

as desired $\square$

We now give the message complexity of algorithm AX.

Theorem 7.18 Algorithm AX solves the asynchronous *Omni-Do*$_{\mathcal{A}_{FM}}(n, p, f)$ problem with message complexity

$$\mathcal{M}_{f_r,f_m} < 4\,(n \cdot f_r + n + p \cdot f_m)\,.$$

Proof: By induction on the number of views, denoted by ℓ, occurring in any execution. For a specific execution ξ_ℓ with ℓ views, let $f_r(\xi_\ell) = f_r^{(\ell)}$ be the fragmentation-number and $f_m(\xi_\ell) = f_m^{(\ell)}$ be the merge-number.

Base Case: $\ell = 0$. Since $f_r^{(\ell)}$ and $f_m^{(\ell)}$ must also be 0, the base case follows from Lemma 7.15.

Inductive Hypothesis: Assume that for all $\ell \leq k$, $\mathcal{M}_{f_r^{(\ell)},f_m^{(\ell)}} < 4(n \cdot f_r^{(\ell)} + n + p \cdot f_m^{(\ell)})$.

Inductive Step: Need to show that for $\ell = k+1$, $\mathcal{M}_{f_r^{(k+1)},f_m^{(k+1)}} < 4(n \cdot f_r^{(k+1)} + n + p \cdot f_m^{(k+1)})$.

Consider a specific execution ξ_{k+1} with $\ell = k + 1$. Let $\Gamma_{\xi_{k+1}}$ be the view-graph induced by this execution. The view-graph has at least one vertex such that all of its descendants are sinks (Fact 7.1). Let ν be such a vertex. We consider two cases:

Case 1: Vertex ν has a descendant μ that corresponds to a merge in the execution. Therefore all ancestors of μ in $\Gamma_{\xi_{k+1}}$ have outdegree 1. Since μ is a sink vertex, the group that corresponds to μ performs all the remaining (if any) tasks and no further messages are sent. Let $\xi_k = \bar{\xi}^{\mu}_{k+1}$ (per Definition 7.6) be an execution in which this merge does not occur. In execution ξ_k, the number of new views is k. Also, $f_r^{(k+1)} = f_r^{(k)}$ and $f_m^{(k+1)} = f_m^{(k)} + 1$. By inductive hypothesis, $\mathcal{M}_{f_r^{(k)}, f_m^{(k)}} < 4(n \cdot f_r^{(k)} + n + p \cdot f_m^{(k)})$. Hence, the message complexity, using Lemma 7.16 is:

$$\begin{aligned}
\mathcal{M}_{f_r^{(k+1)}, f_m^{(k+1)}} &< \mathcal{M}_{f_r^{(k)}, f_m^{(k)}} + 2p \\
&< 4(n \cdot f_r^{(k)} + n + p \cdot f_m^{(k)}) + 2p \\
&= 4(n \cdot f_r^{(k+1)} + n + p \cdot f_m^{(k+1)} - p) + 2p \\
&= 4nf_r^{(k+1)} + 4n + 4pf_m^{(k+1)} - 4p + 2p \\
&\leq 4(n \cdot f_r^{(k+1)} + n + p \cdot f_m^{(k+1)}).
\end{aligned}$$

Case 2: Vertex ν has no descendants that correspond to a merge in the execution. Therefore, the group that corresponds to ν must fragment, say into q groups. These groups correspond to sink vertices in $\Gamma_{\xi_{k+1}}$, thus they perform all of the remaining (if any) tasks and do not send any additional messages. Let $\xi_{k+1-q} = \bar{\xi}^{\nu}_{k+1}$ (per Definition 7.7) be an execution in which the fragmentation does not occur. In the execution ξ_{k+1-q}, the number of new views is $k+1-q \leq k$. Also, $f_r^{(k+1-q)} = f_r^{(k+1)} - q$ and $f_m^{(k+1-q)} = f_m^{(k+1)}$. By inductive hypothesis, $\mathcal{M}_{f_r^{(k+1-q)}, f_m^{(k+1-q)}} < 4(n \cdot f_r^{(k+1-q)} + n + p \cdot f_m^{(k+1-q)})$. From Lemma 7.15, the message cost in each new group caused by a fragmentation is no more than $4n$. Let M_σ be the total number of messages sent in all q groups. Thus, $M_\sigma \leq 4qn$. By Lemma 7.17, the number of messages sent in execution ξ_{k+1}, is less than the number of messages sent in execution ξ_{k+1-q} and the number of messages sent in all q groups. Hence, the message complexity is:

$$\begin{aligned}\mathcal{M}_{f_r^{(k+1)}, f_m^{(k+1)}} &\leq \mathcal{M}_{f_r^{(k+1-q)}, f_m^{(k+1-q)}} + M_\sigma \\ &< 4(n \cdot f_r^{(k+1-q)} + n + p \cdot f_m^{(k+1-q)}) + M_\sigma \\ &= 4(n \cdot f_r^{(k+1)} - qn + n + p \cdot f_m^{(k+1)}) + M_\sigma \\ &\leq 4nf_r^{(k+1)} - 4qn + 4n + 4pf_m^{(k+1)} + 4qn \\ &= 4(n \cdot f_r^{(k+1)} + n + p \cdot f_m^{(k+1)}).\end{aligned}$$

This completes the proof. □

7.1.5.3 Analysis Under Adversary $\mathcal{A}_F$

Algorithm AX solves the *Omni-Do* problem also under patterns of only fragmentations. Observe that $f = f_r$ and $f_m = 0$ for adversary $\mathcal{A}_F$. The following corollary is derived from Theorems 7.14 and 7.18.

Corollary 7.19 Algorithm AX solves the asynchronous *Omni-Do*$_{\mathcal{A}_F}(n, p, f)$ problem with task-oriented work complexity $W_{\mathcal{A}_F}(n, p, f) \leq \min\{n \cdot f + n,\ n \cdot p\}$ and message complexity $M_{\mathcal{A}_F}(n, p, f) < 4(n \cdot f + n)$.

The adversary considered in [32] was not allowed to "fragment" a group into a single group with the same membership. Such fragmentation is allowed by our definition of $\mathcal{A}_F$. In order to compare our results with the results of [32], we define a more restricted adversary $\mathcal{A}'_F$ that is constrained to fragmenting each group into at least 2 groups. Clearly $\mathcal{A}_F$ is more powerful than $\mathcal{A}'_F$, and from Corollary 7.19 we have the following.

Corollary 7.20 Algorithm AX solves the asynchronous *Omni-Do*$_{\mathcal{A}'_F}(n, p, f)$ problem with $W_{\mathcal{A}'_F}(n, p, f) = O(n \cdot f + n)$ and $M_{\mathcal{A}'_F}(n, p, f) = O(n \cdot f + n)$.

In the rest of this section we deal with adversary $\mathcal{A}'_F$. Our definition of the fragmentation-number f is slightly different from the definition of the fragmentation-number f' in [32]. When a group fragments into k groups, f is defined to be equal to k, but f' is defined to be equal to $k-1$. The next Lemma relates f and f'.

Lemma 7.21 $f' < f < 2f'$.

Proof: Assume that k fragmentations occur. Enumerate the fragmentations arbitrarily. Let the number of the new views in the i^{th} fragmentation be f_i. By the definition of f'_i, $f'_i = f_i - 1$. Thus, $f'_i + 1 = f_i$ which implies that $f_i < f'_i + f'_i = 2f'_i$. But $f' = \sum_{i=1}^{k} f'_i$ and $f = \sum_{i=1}^{k} f_i$. Hence, $f < 2f'$. Now observe that, $f' = \sum_{i=1}^{k} f'_i = \sum_{i=1}^{k}(f_i - 1) = \sum_{i=1}^{k} f_i \ - k = f - k$. Therefore $f > f'$. $\square$

In [32] the work is counted in terms of the rounds executed by the processors. In our analysis we count only the number of task executions (including redundancies). However in our algorithm, for as long as any tasks remain undone in a given group, the processors perform the tasks in rounds, except for the last round. Therefore the difference in work complexity for these two algorithms is at most $f \cdot n$. Thus the different definitions of f, f' and work are subsumed in the big-oh analysis, and without substantial variation in the constants. On the other hand, the message complexity of our algorithm, as shown in Corollary 7.20, is substantially better than the at least quadratic message complexity of the algorithm from [32].

7.2 Competitive Analysis of Omni-Do

Given that no algorithm is able to maintain low total work in the presence of network reconfigurations, we pursue competitive analysis of the *Omni-Do* problem. We consider asynchronous message-passing processors under arbitrary regroupings; in particular, we consider the *Omni-Do* problem under adversary $\mathcal{A}_{GR}$ (presented in Section 3.2.2).

Processors in the same group can share their knowledge of completed tasks and, while they remain connected, avoid doing redundant work. The challenge is to avoid redundant work "globally", in the sense that processors should be performing tasks with anticipation of future changes in the network topology. An optimal algorithm, with full knowledge of the future regroupings, can schedule the execution of the tasks in each group in such a way that the overall task-oriented work is the smallest possible, given the particular sequence of regroupings.

As an example, consider the scenario with 3 processors which, starting from isolation, are permitted to proceed synchronously until each has completed $n/2$ tasks; at this point an adversary chooses a pair of processors to merge into a group. It is easy to show that if N_1, N_2, and N_3 are subsets of $[n]$ of size $n/2$, then there is a pair (N_i, N_j) (where $i \neq j$) so that $|N_i \cap N_j| \geq n/6$: in particular, for *any* scheduling algorithm, there is a pair of processors which, if merged at this point, will have $n/6$ duplicated tasks; this pair alone must then expend $n + n/6$ task-oriented work to complete all n tasks. The optimal off-line algorithm that schedules tasks with full knowledge of future merges, of course, accrues only n task-oriented work for the merged pair, as it can arrange for zero overlap. Furthermore, if the adversary partitions the two merged processors immediately after the merge (after allowing the processors to exchanged information about task executions), then the task-oriented work performed by the merged and then partitioned pair is $n + n/3$; the task-oriented work performed by the optimal algorithm remains unchanged, since it terminates at the merge.

To focus on scheduling issues, we assume that processors in a single group work as a single virtual unit; indeed, we treat them as a single asynchronous processor. To this respect, we assume that communication within groups is instantaneous and reliable. We note that the above assumptions can be approximated by group communication services [95], however the task-oriented work of *Omni-Do* algorithms can be negatively affected in large scale wide-area networks [64].

In this section we formulate a simple randomized algorithm, called algorithm RS, and we compare its expected task-oriented work to the task-oriented work of an optimal off-line algorithm which may schedule tasks with full knowledge of future regroupings. In Section 7.2.1 we formally define the notion of competitiveness and we present terminology borrowed from set theory and graph theory that we use in the remainder sections. In Section 7.2.2 we present algorithm RS and in Section 7.2.3 its analysis. Finally, in Section 7.2.4 we present lower bounds on the competitiveness of *Omni-Do* algorithms that show the optimality of algorithm RS.

7.2.1 Preliminaries

As we already mentioned, we consider adversary $\mathcal{A}_{GR}$. That is, we consider computational topologies C that can be expressed as a (p)-DAG (see Section 3.2.2). For the purpose of the analysis of our randomized algorithm (Section 7.2.3) and to provide lower bound results (Section 7.2.4), we require that adversary $\mathcal{A}_{GR}$ also determines the number of tasks that each group is allowed to complete, before it is involved in another regrouping. To this respect, we annotate the number of tasks that the adversary allows to each group to perform on the

(p)-DAGs. In particular, we augment a given (p)-DAG $C = (V, E)$ with a weight function $h : V \to \mathbb{N}$, so that $h(v)$, $v \in V$, is the number of tasks allowed by the adversary for the processors in group $\gamma(v)$ to performed before the next regrouping (recall that γ is a labeling function from V to $2^{[p]} \setminus \{\emptyset\}$ — see "Adversary $\mathcal{A}_{GR}$" in Section 3.2.2). Function h respects the following two conditions: (a) $\forall v \in V,\ h(v) \leq n$, and (b) for any maximal path $(v_1, \ldots, v_k)$ in C, $\sum h(v_i) \geq n$. We refer to each "annotated" (p)-DAG as a (p, n)-DAG. Note that a given (p)-DAG may derive several different (p, n)-DAGs.

To facilitate for a better understanding of the materials presented in the remainder subsections, we give the definition of a (p, n)-DAG along with an example of a (p, n)-DAG.

Definition 7.8 A (p, n)-DAG is a directed acyclic graph $C = (V, E)$ augmented with a weight function $h : V \to \mathbb{N}$ and a labeling $\gamma : V \to 2^{[p]} \setminus \{\emptyset\}$ so that:

1. $\forall v \in V$, $h(v) \leq n$ and for any maximal path $(v_1, \ldots, v_k)$ in C, $\sum h(v_i) \geq n$. (This guarantees that any algorithm terminates during the computation described by the DAG.)

2. γ possesses the following "initial conditions": $[p] = \dot{\bigcup}_{v:\ indegree(v)=0} \gamma(v)$.

3. γ respects the following "conservation law": there is a function $\phi : E \to 2^{[p]} \setminus \{\emptyset\}$ so that for each $v \in V$ with $indegree(v) > 0$, $\gamma(v) = \dot{\bigcup}_{(u,v)\in E} \phi\big((u, v)\big)$,
   and for each $v \in V$ with $outdegree(v) > 0$, $\gamma(v) = \dot{\bigcup}_{(v,u)\in E} \phi\big((v, u)\big)$.

Here $\dot{\cup}$ denotes disjoint union. Finally, for two vertices $u, v \in V$, we write $u \leq v$ if there is a directed path from u to v; we then write $u < v$ if $u \leq v$ and u and v are distinct.

Example 7.4 Consider the $(12, n)$-DAG shown on Figure 7. Here we have $g_1 = \{p_1\}$, $g_2 = \{p_2, p_3, p_4\}$, $g_3 = \{p_5, p_6\}$, $g_4 = \{p_7\}$, $g_5 = \{p_8, p_9, p_{10}, p_{11}, p_{12}\}$, $g_6 = \{p_1, p_2, p_3, p_4, p_6\}$,

$g_7 = \{p_8, p_{10}\}$, $g_8 = \{p_9, p_{11}, p_{12}\}$, $g_9 = \{p_1, p_2, p_3, p_4, p_6, p_8, p_{10}\}$, $g_{10} = \{p_5, p_{11}\}$, and $g_{11} = \{p_9, p_{12}\}$.

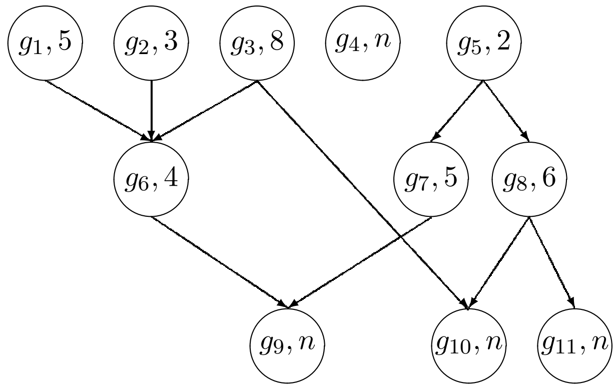

Figure 7: An example of a $(12, n)$-DAG.

This computation template models all (asynchronous) computations with the following behavior: (i) The processors in groups g_1 and g_2 and processor p_6 of group g_3 are regrouped during some regrouping to form group g_6. Processor p_5 of group g_3 becomes a member of group g_{10} during the same regrouping (see below). Prior to this regrouping, processor p_1 (the singleton group g_1) has performed exactly 5 tasks, the processors in g_2 have cooperatively performed exactly 3 tasks and the processors in g_3 have cooperatively performed exactly 8 tasks (assuming that $n > 8$). (ii) Group g_5 is partitioned during some regrouping into two new groups, g_7 and g_8. Prior to this regrouping, the processors in g_5 have performed exactly 2 tasks. (iii) Groups g_6 and g_7 merge during some regrouping and form group g_9. Prior to this merge, the processors in g_6 have performed exactly 4 tasks (counting only the ones performed after the formation of g_6 and assuming that there are at least 4 tasks remaining to be done) and the processors in g_7 have performed exactly 5 tasks. (iv) The processors in group g_8 and processor p_5 of group g_3 are regrouped during some regrouping into groups g_{10} and g_{11}. Prior to this regrouping, the processors in group g_8 have performed exactly 6 tasks (assuming that there are at least 6 tasks remaining, otherwise they would have performed the remaining tasks). (v) The processors in g_9, g_{10}, and g_{11} run until completion with no further regroupings. (vi) Processor p_7 (the singleton group g_4) runs in isolation for the entire computation.

Before we formally define the notion of competitiveness, we introduce some terminology.

Let D be a deterministic algorithm for *Omni-Do* and C a computation template. We let $W_{\mathrm{D}}(C)$ denote the task-oriented work expended by algorithm D, where regroupings are determined according to the computation template C. That is, if $\xi \in \mathcal{E}(\mathrm{D}, \mathcal{A}_{GR})$ is the resulting execution of algorithm D under computation template C, then $W_{\mathrm{D}}(C)$ is the task-oriented work of execution ξ. We let OPT denote the optimal (off-line) algorithm. Specifically, for each C we define $W_{\mathrm{OPT}}(C) = \min_{\mathrm{D}} W_{\mathrm{D}}(C)$.

We treat randomized algorithms as distributions over deterministic algorithms; for a set Z and a family of deterministic algorithms $\{\mathrm{D}_\zeta \mid \zeta \in Z\}$ we let R $= \mathcal{R}(\{\mathrm{D}_\zeta \mid \zeta \in Z\})$ denote the randomized algorithm where ζ is selected uniformly at random from Z and scheduling is done according to D_ζ. For a real-valued random variable X, we let $\mathbb{E}[X]$ denote its expected value. Then,

Definition 7.9 Let α be a real valued function defined on the set of all (p, n)-DAGs (for all p and n). A randomized algorithm R is **α-competitive** if for all computation templates C,

$$\mathbb{E}[W_{\mathrm{D}_\zeta}(C)] \leq \alpha(C) W_{\mathrm{OPT}}(C),$$

this expectation being taken over uniform choice of $\zeta \in Z$.

Note that usually α is fixed for all inputs; we shall see that this would be meaningless in our model. Presently, we use a function α that depends on a certain parameter (see Definition 7.13) of the graph structure of C.

We conclude this subsection with some terminology that we use in the remainder of Section 7.2.

Definition 7.10 A *partially ordered set* or *poset* is a pair $(P, \leq)$ where P is a set and $\leq$ is a binary relation on P for which *(i)* for all $x \in P$, $x \leq x$, *(ii)* if $x \leq y$ and $y \leq x$, then $x = y$, and *(iii)* if $x \leq y$ and $y \leq z$, then $x \leq z$. For a poset $(P, \leq)$ we overload the symbol P, letting it denote both the set and the poset.

Definition 7.11 Let P be a poset. We say that two elements x and y of P are *comparable* if $x \leq y$ or $y \leq x$; otherwise x and y are *incomparable*. A *chain* is a subset of P such that any two elements of this subset are comparable. An *antichain* is a subset of P such that any two distinct elements of this subset are incomparable. The *width* of P, denoted $\mathbf{w}(P)$, is the size of the largest antichain of P.

Associated with any directed acyclic graph (DAG) $C = (V, E)$ is the natural *vertex poset* $(V, \leq)$ where $u \leq v$ if and only if there is a directed path from u to v. Then the *width of* C, denoted $\mathbf{w}(C)$, is the width of the poset $(V, \leq)$.

Definition 7.12 Given a DAG $C = (V, E)$ and a vertex $v \in V$, we define the *predecessor graph at* v, denoted $P_C(v)$, to be the subgraph of C that is formed by the union of all paths in C terminating at v. Likewise, the *successor graph at* v, denoted $S_C(v)$, is the subgraph of C that is formed by the union of all the paths in C originating at v.

In Section 3.2.2 we informally defined the notion of the *computation width* of a computation template (that is, of a (p, n)-DAG)). We now give its formal definition.

Definition 7.13 The *computation width* of a DAG $C = (V, E)$, denoted $\mathbf{cw}(C)$, is defined as

$$\mathbf{cw}(C) = \max_{v \in V} \mathbf{w}(S_C(v)).$$

Note that the processors that comprise a group formed during a computation template C may be involved in many different groups at later stages of the computation, but no more than $\mathbf{cw}(C)$ of these groups can be computing in ignorance of each other's progress.

Example 7.5 In the $(12, n)$-DAG of Figure 7, the maximum width among all successor graphs is 3: $\mathbf{w}(S((g_5, 2))) = 3$. Therefore, the computation width of this DAG is 3. Note that the width of the DAG is 6 (nodes $(g_1, 5), (g_2, 3), (g_3, 8), (g_4, n), (g_7, 5)$ and $(g_8, 6)$ form an antichain of maximum size).

7.2.2 Description of Algorithm RS

We consider the natural randomized algorithm RS where a processor (or group) with knowledge that the tasks τ in a set $K \subset [n]$ have been completed selects to next complete a task at random from the set $[n] \setminus K$. (Recall that we treat randomized alorithms as distributions over deterministic algorithms.) More formally, let $\Pi = (\pi_1, \ldots, \pi_p)$ be a p-tuple of permutations, where each π_i is a permutation of $[n]$. We describe a deterministic algorithm D_Π so that

$$\mathrm{RS} = \mathcal{R}\big(\{\mathrm{D}_\Pi \mid \Pi \in (S_n)^p\}\big);$$

here S_n is the collection of permutations on $[n]$. Let G be a group of processors and $q \in G$ the processor in G with the lowest processor identifier. Then the deterministic algorithm D_Π specifies that the group G, should it know that the tasks in $K \subset [n]$ have been completed, next completes the first task in the sequence $\pi_q(1), \ldots, \pi_q(n)$ which is not in K.

7.2.3 Analysis of Algorithm RS

We now analyze the competitive ratio (in terms of task-oriented work) of algorithm RS. For algorithm RS subjected to a computation template C we write $W_{\mathrm{RS}}(C) = \mathbb{E}\left[W_{\mathrm{RS}}(C)\right]$, this expectation taken over the random choices of the algorithm. Where C can be inferred from context, we simply write W_{RS} and W_{OPT}.

We first recall Dilworth's Lemma [29], a duality theorem for posets:

Lemma 7.22 [29] The width of a poset P is equal to the minimum number of chains needed to cover P. (A family of nonempty subsets of a set Q is said to *cover* Q if their union is Q.)

We will also use a generalized degree-counting argument:

Lemma 7.23 Let $G = (U, V, E)$ be an undirected bipartite graph with no isolated vertices and $h : V \to \mathbb{R}$ a non-negative weight function on G. For a vertex v, let $\Gamma(v)$ denote the vertices adjacent to v. Suppose that for some $B_1 > 0$ and for each vertex $u \in U$ we have $\sum_{v\in\Gamma(u)} h(v) \leq B_1$ and that for some $B_2 > 0$ and for each vertex $v \in V$ we have $\sum_{u\in\Gamma(v)} h(u) \geq B_2$, then $\dfrac{\sum_{u\in U} h(u)}{\sum_{v\in V} h(v)} \geq \dfrac{B_2}{B_1}$.

Proof: We compute the quantity $\sum_{(u,v)\in E} h(u)h(v)$ by expanding according to each side of the bipartition:

$$B_1 \sum_{u\in U} h(u) \geq \sum_{u\in U}\Big(h(u)\cdot \sum_{v\in\Gamma(u)} h(v)\Big) = \sum_{(u,v)\in E} h(u)h(v) = \sum_{v\in V}\Big(h(v)\cdot \sum_{u\in\Gamma(v)} h(u)\Big) \geq B_2 \sum_{v\in V} h(v).$$

As $B_1 > 0$ and $\sum_v h(v) \geq B_2 > 0$, we conclude that $\dfrac{\sum_{u\in U} h(u)}{\sum_{v\in V} h(v)} \geq \dfrac{B_2}{B_1}$, as desired. □

We now establish an upper bound on the competitive ratio of the algorithm RS.

Theorem 7.24 Algorithm RS is $(1 + \mathbf{cw}(C)/e)$-competitive for any (p, n)-DAG $C = (V, E)$.

Proof: Let C be a (p, n)-DAG; recall that associated with C are the two functions $h : V \to \mathbb{N}$ and $\gamma : V \to 2^{[p]} \setminus \{\emptyset\}$. For a subgraph $C' = (V', E')$ of C, we let $H(C') = \sum_{v\in V'} h(v)$. Recall that $P_C(v)$ and $S_C(v)$ denote the predecessor and successor graphs of C at v. Then, we say that a vertex $v \in V$ is *saturated* if $H(P_C(v)) \le n$; otherwise, v is *unsaturated*. Note that if v is saturated, then the group $\gamma(v)$ must complete $h(v)$ tasks *regardless of the scheduling algorithm used*. Along these same lines, if v is an unsaturated vertex for which $n > \sum_{u<v} h(u)$, the group $\gamma(v)$ must complete at least $\max(h(v), n - \sum_{u<v} h(u))$ tasks under any scheduling algorithm. As these portions of C which correspond to computation which must be performed by any algorithm will play a special role in the analysis, it will be convenient for us to rearrange the DAG so that all such work appears on saturated vertices. To achieve this, note that if v is an unsaturated vertex for which $\sum_{u<v} h(u) < n$, we may replace v with a pair of vertices, v_s and v_u, where all edges directed into v are redirected to v_s, all edges directed out of v are changed to originate at v_u, the edge (v_s, v_u) is added to E, and h is redefined so that

$$h(v_s) = n - \sum_{u<v} h(u) \qquad \text{and} \qquad h(v_u) = h(v) - h(v_s).$$

Note that the graph C' obtained by altering C in this way corresponds to the same computation, in the sense that $W_{\mathrm{D}}(C) = W_{\mathrm{D}}(C')$ for any algorithm D. For the remainder of the proof we will assume that this alteration has been made at every relevant vertex, so that the graph C satisfies the condition

$$v \text{ unsaturated } \Rightarrow \sum_{u<v} h(u) \ge n. \tag{2}$$

Finally, for a vertex v, we let T_v be the random variable equal to the number of tasks that RS completes at vertex v. Note that if v is saturated, then $T_v = h(v)$. Let $\mathcal{S}$ and $\mathcal{U}$ denote the sets of saturated and unsaturated vertices, respectively. Given the above definitions, we immediately have

$$W_{\mathrm{OPT}} \ge \sum_{s \in \mathcal{S}} h(s)$$

and, by linearity of expectation,

$$W_{\mathrm{RS}} = \mathbb{E}\Big[\sum_v T_v\Big] = \sum_{s \in \mathcal{S}} h(s) + \sum_{u \in \mathcal{U}} \mathbb{E}[T_u] \le W_{\mathrm{OPT}} + \sum_{u \in \mathcal{U}} \mathbb{E}[T_u]. \tag{3}$$

Our goal is to conclude that for some appropriate β,

$$\mathbb{E}\left[\sum_{u\in\mathcal{U}} T_u\right] \leq \beta \cdot \sum_{s\in\mathcal{S}} h(s) \leq \beta \cdot W_{\text{OPT}}$$

and hence that RS is $1+\beta$ competitive. We will obtain such a bound by applying Lemma 7.23 to an appropriate bipartite graph, constructed next.

Given $C = (V, E)$ construct the (undirected) bipartite graph $G = (\mathcal{S}, \mathcal{U}, E_G)$ where $E_G = \{(s, u) \mid s < u\}$. As in Lemma 7.23, for a vertex v, we let $\Gamma(v)$ denote the set of vertices adjacent to v. Now assign weights to the vertices of G according to the rule $h^*(v) = \mathbb{E}[T_v]$. Note that for $s \in \mathcal{S}, h^*(s) = h(s)$ and hence by condition (2) above, we immediately have the bound

$$\forall u \in \mathcal{U}, \quad \sum_{s\in\Gamma(u)} h^*(s) \geq n. \tag{4}$$

We now show that $\forall s \in \mathcal{S}$,

$$\sum_{u\in\Gamma(s)} h^*(u) \leq \mathbf{cw}(C) \cdot \frac{n}{e}. \tag{5}$$

Before proceeding to establish this bound, note that equations (4) and (5), together with Lemma 7.23 imply that

$$W_{\text{RS}}(C) \quad \leq \quad \sum_{s\in\mathcal{S}} h(s) + \sum_{u\in\mathcal{U}} h^*(u) \leq \Big(1 + \frac{\mathbf{cw}(C)}{e}\Big) \sum_{s\in\mathcal{S}} h(s) \leq \Big(1 + \frac{\mathbf{cw}(C)}{e}\Big) W_{\text{OPT}}(C),$$

as desired.

Returning now to equation (5), let $s \in \mathcal{S}$ be a saturated vertex and consider the successor graph (of C) at s, $S_C(s)$. By Lemma 7.22 (Dilworth's Lemma), there exist $w \triangleq \mathbf{w}(S_C(s)) \leq \mathbf{cw}(C)$ paths in $S_C(s)$, $P_1, P_2, \ldots P_w$ so that their union covers $S_C(s)$. Let X_i be the random variable whose value is the number of tasks performed by RS on the portion of the path P_i consisting of unsaturated vertices. Note that if $u \in V$ is unsaturated and $u \leq v$, then v is unsaturated and hence, for each path P_i, there is a first unsaturated vertex u_i^0 after which every vertex of P_i is unsaturated. Note now that for a fixed individual task τ, conditioned upon the event that τ is not yet complete, the probability that τ is *not* chosen by RS for completion at a given selection point in $P_C(u_i^0)$ is no more than $(1 - 1/n)$. Let L_i be the random variable whose value is the set of tasks left incomplete by RS at the formation of the group $\gamma(u_i^0)$. As u_i^0 is unsaturated, $\sum_{v<u_i^0} h(v) \geq n$ by condition (2) and hence, for each i,

$$\Pr[\tau \in L_i] \leq (1 - 1/n)^n \leq 1/e.$$

As there are a total of n tasks,

$$\mathbb{E}[|L_i|] \leq n/e.$$

Of course, since RS completes a new task at each step, $X_i \leq |L_i|$ so that $\mathbb{E}[X_i] \leq n/e$ and by the linearity of expectation

$$\mathbb{E}\Big[\sum_i X_i\Big] \leq w \cdot n/e.$$

Now every unsaturated vertex in $S_C(s)$ appears in some P_i and hence

$$\sum_{u\in\Gamma(s)} h^*(u) \leq \mathbb{E}\Big[\sum_i X_i\Big] \leq wn/e \leq \mathbf{cw}(C) \cdot n/e,$$

as desired. □

7.2.4 Lower Bounds

We now show that the competitive ratio achieved by algorithm RS is tight. We begin with a lower bound for *deterministic* algorithms. This is then applied to give a lower bound for randomized algorithms in Corollary 7.26.

Theorem 7.25 Let $a : \mathbb{N} \to \mathbb{R}$ and D be a deterministic algorithm for *Omni-Do* so that D is $a(\mathbf{cw}(\cdot))$-competitive (that is D is α-competitive, for a function $\alpha = a \circ \mathbf{cw}$)). Then $a(c) \geq 1 + c/e$.

Proof: Fix $k \in \mathbb{N}$. Consider the case when $n = p = g \gg k$ and $n \bmod k = 0$, g being the number of initial groups. We consider a computation template $C_{\mathbf{G}}$ determined by a tuple $\mathbf{G} = (G_1, \ldots, G_{n/k})$ where each $G_i \subset [n]$ is a set of size k and $\bigcup_i G_i = [n]$. Initially, the computation template $C_{\mathbf{G}}$ has the processors synchronously proceed until each has completed n/k tasks; at this point, the processors in G_i are merged and allowed to exchange information about task executions. Each G_i is then immediately partitioned into c groups. Note that the off-line optimal algorithm accrues exactly n^2/k work for this computation template (it terminates prior to the partitions of the G_i).

We will show that for any D, there is a selection of the G_i so that

$$W_{\mathrm{D}}(C_{\mathbf{G}}) \geq n^2/k\left[1 + c(1-\frac{1}{k})^k - o(1)\right],$$

and hence that $a(c) \geq 1 + c/e$. Consider the behavior of D when the $\mathbf{G}$ is selected at random, uniformly among all such tuples. Let $P_i \subset [n]$ be the subset of n/k tasks completed by processor i before the merges take place; these sets are determined by the algorithm D. We begin by bounding

$$\mathop{\mathbb{E}}_{\mathbf{G}}\left[\Big|\bigcup_{i\in G_1} P_i\Big|\right].$$

To this end, consider an experiment where we select k sets $Q_1, \ldots, Q_k$, each Q_i selected independently and uniformly from the set $\{P_i\}$. Now, for a specific task τ, let $p_\tau = \Pr_{Q_1}[\tau \notin Q_1]$, so that $\Pr_{Q_i}[\tau \notin \bigcup_i Q_i] = p_\tau^k$. As the Q_i are selected independently,

$$\mathop{\mathbb{E}}_{Q_i}\Big[\big|[n] - \bigcup_i Q_i\big|\Big] = \sum_\tau p_\tau^k.$$

Observe now that

$$\sum_\tau (1-p_\tau) = \sum_\tau \Pr_{Q_1}[\tau \in Q_1] = \mathop{\mathbb{E}}_{Q_1}[|Q_1|] = n/k$$

and hence $\sum_\tau p_\tau = n(1-1/k)$. As the function $x \mapsto x^k$ is convex on $[0,\infty)$, $\sum_\tau p_\tau^k$ is minimized when the p_τ are equal and we must have

$$\mathop{\mathbb{E}}_{Q_i}\Big[\big|[n] - \bigcup_i Q_i\big|\Big] \geq n \cdot \left(1 - \frac{1}{k}\right)^k.$$

Now observe that, conditioned on the Q_i being distinct, the distribution of $(Q_1, \ldots, Q_k)$ is identical to that of $(P_{g_1^1}, \ldots, P_{g_k^1})$ where the random variable $G_1 = \{g_1^1, \ldots, g_k^1\}$. Considering that $\Pr[\exists i \neq j, Q_i = Q_j] \leq k^2/n$, we have

$$\mathop{\mathbb{E}}_{Q_i}\Big[\big|[n] - \bigcup_i Q_i\big|\Big] \leq \left(1 - \frac{k^2}{n}\right) \mathop{\mathbb{E}}_{\mathbf{G}}\Big[n - \big|\bigcup_{i \in G_1} P_i\big|\Big] + 1 \cdot \frac{k^2}{n}$$

and hence as $n \to \infty$ we see that the expected number of tasks remaining for those processors in group G_1 is

$$\mathop{\mathbb{E}}_{\mathbf{G}}\left[n - \Big|\bigcup_{i \in G_1} P_i\Big|\right] \geq n(1 - 1/k)^k - o(1).$$

Of course, the distribution of each G_i is the same, so that

$$\mathop{\mathbb{E}}_{\mathbf{G}}\left[\sum_{i=1}^{n/k}\Big(n - \big|\bigcup_{j \in G_i} P_j\big|\Big)\right] = [1 - o(1)]\left(\frac{n}{k}\right) \cdot n\left(1 - \frac{1}{k}\right)^k.$$

In particular, there must exist a specific selection of $\mathbf{G} = (G_1, \ldots, G_{n/k})$ which achieves this bound. Recall that every G_i is partitioned into c groups. Therefore, for such $\mathbf{G}$, the total work is at least

$$\frac{n^2}{k} \cdot \left(1 + [1 - o(1)] \cdot c \cdot (1 - \frac{1}{k})^k\right).$$

As $\lim_{k \to \infty}(1 - \frac{1}{k})^k = \frac{1}{e}$, this completes the proof. □

As the above stochastic computation template $C_{\mathbf{G}}$ is independent of the deterministic algorithm D, this immediately gives rise to a lower bound for randomized algorithms:

Corollary 7.26 Let $a : \mathbb{N} \to \mathbb{R}$ and $\mathcal{R}\big(\{\mathrm{D}_\zeta \mid \zeta \in Z\}\big)$ be a randomized algorithm for *Omni-Do* that is $(a \circ \mathbf{cw})$-competitive. Then $a(c) \geq 1 + c/e$.

Proof: Assume for contradiction that for some c, $a(c) < 1 + c/e$ and let k be large enough so that $(1 - \frac{1}{k})^k > a(c) - 1$. For this k we proceed as in the proof above, considering a random $\mathbf{G}$ and the computation template $C_{\mathbf{G}}$ with $n = g = p$ congruent to $0 \bmod k$, g being the number of initial groups. Then, as above,

$$\begin{aligned} \mathop{\mathbb{E}}_{\mathbf{G}}\left[\mathop{\mathbb{E}}_{\zeta}\left[W_{\mathrm{D}_\zeta}(C_{\mathbf{G}})\right]\right] &= \mathop{\mathbb{E}}_{\zeta}\left[\mathop{\mathbb{E}}_{\mathbf{G}}\left[W_{\mathrm{D}_\zeta}(C_{\mathbf{G}})\right]\right] \geq \min_{\zeta}\left[\mathop{\mathbb{E}}_{\mathbf{G}}\left[W_{\mathrm{D}_\zeta}(C_{\mathbf{G}})\right]\right] \\ &\geq \frac{n^2}{k} \cdot \left(1 + [1 - o(1)] \cdot c \cdot (1 - \frac{1}{k})^k\right). \end{aligned}$$

Hence there exists a $\mathbf{G}$ so that $\mathbb{E}_\zeta\left[W_{\mathrm{D}_\zeta}(C_{\mathbf{G}})\right] \geq \frac{n^2}{k} \cdot \left(1 + [1 - o(1)]\frac{c}{e}\right)$, which completes the proof. □

The above result yields the optimality of algorithm RS. Specifically, RS achieves the optimal competitive ratio over the set of all computation templates with a given computation width.

Chapter 8

Conclusions and Future Work

This thesis studies the impact of the adverse environment on the efficiency of distributed cooperative computing. In particular, the thesis considers the *Do-All* problem where p processors must cooperatively perform n tasks in the presence of adversity, and develops upper and lower bound results that demonstrate precisely how adversity affects *Do-All* solutions. We summarize the contributions of the thesis and discuss future research directions.

The thesis presents *Do-All* lower bounds on work for synchronous crash-prone processors that capture the dependence of work not only on n and p, but also on f, the number of crashes, for the entire range of f ($1 \leq f < p$). This gives the first non-trivial lower bound for *Write-All* work for a moderate number of failures ($f \leq p/\log p$). For the model of computation where processors are able to make perfect load-balancing decisions locally (the perfect knowledge assumption), matching upper bounds are given. An important contribution of the thesis is the definition of the *iterative Do-All* problem that models the repetitive use of *Do-All* algorithms, such as found in algorithm simulations, and the development of failure-sensitive bounds for r-*iterative Do-All* work, that are stronger than the r-fold work complexity of a single *Do-All*. The thesis introduces an approach where the analysis of specific algorithms can be divided into two parts: (i) the analysis of the cost of tolerating failures while assuming "free" load-balancing, and (ii) the analysis of the cost of implementing load-balancing. The utility and generality of this approach is demonstrated by deriving new failure-sensitive analysis of three known efficient algorithms: algorithm W (for the synchronous shared-memory model), algorithm KMS (for the synchronous shared-memory model with controlled memory access concurrency), and algorithm AN [17] (for the synchronous message-passing model). For each of the three algorithms, substantial improvement in the analysis is recorded, especially for a moderate number of failures ($f \leq p/\log p$). Also, by iteratively using algorithms W, KMS, and AN and using the new approach to their failure-sensitive analyses, we obtain tighter upper bounds for the *iterative Write-All* problem in shared-memory systems, and the first non-trivial upper bound analysis of the *iterative Do-All* problem in message-passing systems.

An interesting research direction is to develop failure-sensitive upper and lower bounds on the work of *Do-All* for the model with processor crashes and restarts. As mentioned in Section 2.3, the prior bounds for *Do-All* under the assumption of perfect knowledge for this setting are not failure-sensitive [68] (both upper and lower bounds are given as functions of only n and p). Also, the bounds on work given for *Do-All* in the message-passing and shared-memory models for processor crashes and restarts do not adequately show the dependence of work on the crashes and restarts (see Sections 2.1 and 2.2). A possible direction toward this is to investigate whether the approach used in the model with processor crashes can also be successfully applied here: given an algorithm, first analyze the cost of tolerating crashes and restarts assuming perfect load-balancing, and then analyze the cost of implementing perfect load-balancing based on the structure of the algorithm. The challenge here is to overcome the additional complication resulted by the ability of processors to restart after crashing.

Another contribution of the thesis is the development of a new robust algorithm for p synchronous processors that solves the *Do-All* problem with n tasks in the presence of any pattern of f crashes ($f < p$). This algorithm achieves asymptotically better work complexity than the algorithm of Galil, Mayer, and Yung [44] (the previously best known algorithm for this setting) while obtaining the same message complexity. Unlike algorithm AN [17] that has comparable work complexity (even using our new failure-sensitive analysis) but uses reliable multicast, the new algorithm uses simple point-to-point messaging. The algorithm uses an approach where processors share information using a new gossip algorithm. The processors decide where to send a gossip message based on sets of permutations with special combinatorial properties that we show to exist. This gossip algorithm achieves substantially better message complexity than the message complexity of the previously best known gossip algorithm of Chlebus and Kowalski [21], while obtaining the same asymptotic time complexity.

Both our *Gossip* and *Do-All* algorithms work correctly under any set of permutations, but the complexity result can only be guaranteed under the permutations with specific combinatorial properties that we show only to exist. A future direction is to investigate how to efficiently construct these permutations. Another direction is to extend the technique of using a gossip algorithm for information sharing to the model with synchronous restartable crash-prone processors and develop an efficient algorithm that solves *Do-All* using point-to-point messaging. (Recall that algorithm AR [17] is the only known algorithm that efficiently solves *Do-All* for synchronous restartable crash-prone processors, but it does so under the strong assumption of reliable multicast.) This gives rise to another interesting research problem: how is the *Gossip* problem formulated in the presence of crashes and restarts? The challenge is to specify the termination condition: When should the problem be considered as solved? In the presence of only processor crashes, the problem is considered solved when each non-faulty processor either knows the rumor of a processor or it knows that the processor crashed. This is no longer sufficient for the case of processor crashes and restarts.

The thesis substantially contributes to the study of the *Omni-Do* problem in partitionable networks, where algorithms must deal with groups of processors that become disconnected and reconnected during the computation. The thesis presents a new robust algorithm, called algorithm AX, that solves *Omni-Do* for asynchronous processors under group fragmentations and merges. This extends the work of Dolev, Segala and Shvartasman [32], that considers only group fragmentations. In addition, algorithm AX has better message complexity (subquadratic in n) than the algorithm of Dolev *et al.* (at least quadratic in n) and the same task-oriented work complexity under group fragmentations. Algorithm AX relies on a group communication service (GCS) [95] with certain properties to provide membership and communication services. These properties are basic and are provided by several group communication systems and specifications [23]. For the analysis of the algorithm, the notion of *view-graphs* is introduced. View-graphs are directed acyclic graphs used to represent the partially-ordered view evolution history witnessed by the processors. We believe that view-graphs have the potential of serving as a general tool for studying cooperative computing with group communication services.

A recent study performed by Jacobsen, Zhang, and Marzullo [64] demonstrated that algorithm AX may not be practical in wide-area networks. In particular, they showed, via trace analysis, that algorithm AX performs poorly with respect to the total completion time. The authors argue that the reason for this is the use of group communication services that do not scale well in large networks, where communication is less likely to be transitive and symmetric (as assumed by group communications). They substantiate this argument by simulating algorithm

AX in a wide-area network and comparing its performance with that of a simpler algorithm. The simpler algorithm, which has much larger worst case task-oriented work complexity than AX, appears to work much better in practice. That algorithm does not use group communication services, but instead it uses a technique that relies on leases [56]. However, as the authors point out, group communications can be used effectively in LANs. Thus it is interesting to evaluate the performance of AX in LANs.

Given that no algorithm is able to maintain low total worst case task-oriented work in the presence of network partitions, the thesis initiates the study of *Omni-Do* as an on-line problem and pursues competitive analysis. Specifically, a simple randomized algorithm, called algorithm RS, is introduced and analyzed under arbitrary patterns of network reconfigurations. The thesis establishes bounds on the competitive ratio of algorithm RS and shows that for the relevant gradation of the computation templates these bounds are tight, by proving lower bounds. These results lead to a better understanding on the effectiveness of *Omni-Do* computations in partitionable networks and demonstrate precisely the impact of partitions on the efficiency of the computation.

One outstanding problem is to derandomize the schedules used by task-performing algorithms and produce task-oriented work- and message-competitive *deterministic* algorithms for *Omni-Do*. Another promising direction is to study the task-performing paradigm in models of computation that combine network reconfigurations with processor failures. The goal is to establish complexity results that show how the performance of task-performing algorithms depends on both on the extent of the network reconfiguration and on the number of processor failures.

The thesis has considered the *Do-All* problem under the assumption that the number of participating processors p and the number of tasks n is fixed, bounded, and known *a priori*. It would be equally important to consider *Do-All* in dynamic systems, where the number of processors and tasks are not known and are not bounded. The *Do-All* problem in such settings abstracts *web-based computing* (see section 2.8), where a large number of processing elements cooperate via the Internet in computing a large number of independent tasks (e.g., SETI [74]) that a fixed-size collection of processing machines would not be able to handle. The set of processing elements available to the computation may dynamically change, possibly due to processor failures or processors becoming unavailable during periods when they are required to perform other unrelated (local) computations, or due to repaired or idle processors joining the computation already in progress. Furthermore, tasks are generated dynamically and different tasks may be known to different processors. Developing algorithms for *Do-All* in such dynamic systems is very challenging, since these algorithms must not only tolerate component failures, but they must also deal with the dynamic nature of the system. The *Do-All* problem must be formulated for such settings, and new efficiency measures need to be defined, since the established measures of efficiency assume that the number of tasks and the number of processors are known. One approach to evaluating *Do-All* algorithms in dynamic systems is to express the measures of efficiency as functions of time. Ongoing research is attempting to formulate a theoretical framework, that would enable the study of the *Do-All* problem in dynamic systems.

Bibliography

[1] M. Abdelguerfi and S. Lavington. *Emerging Trends in Database and Knowledge-Base Machines: The Application of Parallel Architectures to Smart Information Systems*. IEEE Press, 1995.

[2] C. Aguirre, J. Martinez-Munoz, F. Corbacho, and R. Huerta. Small-world topology for multi-agent collaboration. In *Proceedings of the* 11^{th} *International Workshop on Database and Expert Systems Applications*, pages 231–235, 2000.

[3] M. Ajtai, J. Aspnes, C. Dwork, and O. Waarts. A theory of competitive analysis for distributed algorithms. In *Proceedings of the* 35^{th} *Symposium on Foundations of Computer Science (FOCS 1994)*, pages 401–411, 1994.

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

[5] N. Alon, H. Kaplan, M. Krivelevich, D. Malkhi, and J. Stern. Scalable secure storage when half the system is faulty. In *Proceedings of the* 27^{th} *International Colloquium on Automata, Languages and Programming (ICALP 2000)*, pages 577–587, 2000.

[6] N. Alon and J.H. Spencer. *The Probabilistic Method*. J. Wiley and Sons, Inc., second edition, 2000.

[7] R.J. Anderson and H. Woll. Algorithms for the certified Write-All problem. *SIAM Journal of Computing*, 26(5):1277–1283, 1997.

[8] J. Aspnes and W. Hurwood. Spreading rumors rapidly despite an adversary. *Journal of Algorithms*, 26(2):386–411, 1998.

[9] Y. Aumann and M.O. Rabin. Clock construction in fully asynchronous parallel systems and PRAM simulation. In *Proceedings of the* 33^{rd} *IEEE Symposium on Foundations of Computer Science (FOCS 1992)*, pages 147–156, 1992.

[10] O. Babaoglu, R. Davoli, L. Giachini, and M. Baker. Relacs: A communication infrastructure for constructing reliable applications in large-scale distributed systems. In *Proceedings of the* 28^{th} *Hawaii International Conference on System Science (HICSS 1995)*, pages 612–621, 1995.

[11] O. Babaoglu, R. Davoli, and A. Montresor. Group communication in partitionalbe systems: Specification and algorithms. Technical Report UBLCS98-01, Dept. of Computer Science, University of Bologna, 1998.

[12] O. Babaoglu, R. Davoli, A. Montresor, and R. Segala. System support for partition-aware network applications. In *Proceedings of the* 18^{th} *IEEE International Conference on Distributed Computing Systems (ICDCS 1998)*, pages 184–191, 1998.

[13] P. Berman and J. Garay. Cloture voting: $(n/4)$-resilient distributed consensus in $t+1$ rounds. *Mathematical Systems Theory*, 26(1):3–20, 1993.

[14] K.P. Birman and R. van Renesse. *Reliable Distributed Computing with the Isis Toolkit*. IEEE Computer Society Press, 1994.

[15] J. Buss, P.C. Kanellakis, P. Ragde, and A.A. Shvartsman. Parallel algorithms with processor failures and delays. *Journal of Algorithms*, 20(1):45–86, 1996.

[16] T.D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, 1996.

[17] B. Chlebus, R. De Prisco, and A.A. Shvartsman. Performing tasks on restartable message-passing processors. *Distributed Computing*, 14(1):49–64, 2001.

[18] B. Chlebus, S. Dobrev, D. Kowalski, G. Malewicz, A.A. Shvartsman, and I. Vrto. Towards practical deterministic Write-All algorithms. In *Proceedings of the* 13^{th} *ACM Symposium on Parallel Algorithms and Architectures (SPAA 2001)*, pages 271–280, 2001.

[19] B.S. Chlebus, L. Gasieniec, D.R. Kowalski, and A.A. Shvartsman. Bounding work and communication in robust cooperative computation. In *Proceedings of the* 16^{th} *International Symposium on Distributed Computing (DISC 2002)*, pages 295–310, 2002.

[20] B.S. Chlebus and D. R. Kowalski. Randomization helps to perform tasks on processors prone to failures. In *Proceedings of the* 13^{th} *International Symposium on Distributed Computing (DISC 1999)*, pages 284–296, 1999.

[21] B.S. Chlebus and D.R. Kowalski. Gossiping to reach consensus. In *Proceedings of the* 14^{th} *ACM Symposium on Parallel Algorithms and Architectures (SPAA 2002)*, pages 220–229, 2002.

[22] B.S. Chlebus, D.R. Kowalski, and A. Lingas. The Do-All problem in broadcast networks. In *Proceedings of the* 20^{th} *ACM Symposium on Principles of Distributed Computing (PODC 2001)*, pages 117–126, 2001.

[23] G.V. Chockler, I. Keidar, and R. Vitenberg. Group communication specifications: A comprehensive study. *ACM Computing Surveys*, 33(4):1–43, 2001.

[24] F. Cristian. Group, majority and strict agreement in timed asynchronous distributed systems. In *Proceedings of the* 26^{th} *Conference on Fault-Tolerant Computer Systems*, pages 178–187, 1996.

[25] P. Dasgupta, Z. Kedem, and M. Rabin. Parallel processing on networks of workstation: A fault-tolerant, high performance approach. In *Proceedings of the* 15^{th} *IEEE International Conference on Distributed Computer Systems (ICDCS 1995)*, pages 467–474, 1995.

[26] H. Davenport. *Multicative Number Theory*. Springer, second edition, 1980.

[27] R. De Prisco, A. Fekete, N. Lynch, and A.A. Shvartsman. A dynamic view-oriented group communication service. In *Proceedings of the* 17^{th} *ACM Symposium on Principles of Distributed Computing (PODC 1998)*, pages 227–236, 1998.

[28] R. De Prisco, A. Mayer, and M. Yung. Time-optimal message-efficient work performance in the presence of faults. In *Proceedings of the* 13^{th} *ACM Symposium on Principles of Distributed Computing (PODC 1994)*, pages 161–172, 1994.

[29] R.P. Dilworth. A decomposition theorem for partially ordered sets. *Annals of Mathematics*, 51:161–166, 1950.

[30] D. Dolev and D. Malki. The transis approach to high availability cluster communications. *Communications of the ACM*, 39(4):64–70, 1996.

[31] D. Dolev, D. Malki, and R. Strong. A framework for partitionable membership service. Technical Report TR 95-4, Institute of Computer Science, The Hebrew University of Jerusalem, 1995.

[32] S. Dolev, R. Segala, and A.A. Shvartsman. Dynamic load balancing with group communication. In *Proceedings of the* 6^{th} *International Colloquium on Structural Information and Communication Complexity (SIROCCO 1999)*, pages 111–125, 1999.

[33] C. Dwork, J. Halpern, and O. Waarts. Performing work efficiently in the presence of faults. *SIAM Journal on Computing*, 27(5):1457–1491, 1998. A preliminary version appears in the *Proceedings of the* 11^{th} *ACM Symposium on Principles of Distributed Computing (PODC 1992)*, pages 91–102, 1992.

[34] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, 1988.

[35] C. Dwork and Y. Moses. Knowledge and common knowledge in a Byzantine environment: Crash failures. *Information and Computation*, 88(2):156–186, 1990.

[36] R. Elmasri and S.B. Navathe. *Fundamentals of Database Systems*. Addison-Wesley publishing company, second edition, 1994.

[37] P. Ezhilchelvan, R. Macedo, and S. Shrivastava. Newtop: A fault-tolerant group communication protocol. In *Proceedings of the* 15^{th} *IEEE International Conference on Distributed Computing Systems (ICDCS 1995)*, pages 296–306, 1995.

[38] A. Fekete, N. Lynch, and A.A. Shvartsman. Specifying and using a partitionable group communication service. In *Proceedings of the* 16^{th} *ACM Symposium on Principles of Distributed Computing (PODC 1997)*, pages 53–62, 1997.

[39] A. Fekete, N. Lynch, and A.A. Shvartsman. Specifying and using a partitionable group communication service. *ACM Transactions on Computer Systems*, 19(2):171–216, 2001.

[40] M.J. Fischer and N.A. Lynch. A lower bound for the time to assure interactive consistency. *Information Processing Letters*, 14(4):183–186, 1982.

[41] M.J. Fischer, N.A. Lynch, and M.S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, 1985.

[42] J.D. Foley, A. van Dam, S.K. Feiner, and J.F. Hughes. *Computer Graphics: Principle and Practice*. Addison-Wesley publishing company, second edition, 1996.

[43] S. Fortune and J. Wyllie. Parallelism in random access machines. In *Proceedings of the* 10^{th} *ACM Symposium on Theory of Computing (STOC 1978)*, pages 114–118, 1978.

[44] Z. Galil, A. Mayer, and M. Yung. Resolving message complexity of byzantine agreement and beyond. In *Proceedings of the* 36^{th} *IEEE Symposium on Foundations of Computer Science (FOCS 1995)*, pages 724–733, 1995.

[45] G.R. Gallager. A perspective on multi-access channels. *IEEE Transactions on Information Theory*, 31(2):124–142, 1985.

[46] J.A. Garay and Y. Moses. Fully polynomial Byzantine agreement for processors in rounds. *SIAM Journal on Computing*, 27(1):247–290, 1998.

[47] Ch. Georgiou, D.R. Kowalski, and A.A. Shvartsman. Efficient gossip and robust distributed computation. In *Proceedings of the* 17^{th} *International Symposium on Distributed Computing (DISC 2003)*, pages 224–238, 2003.

[48] Ch. Georgiou, A. Russell, and A.A. Shvartsman. The complexity of synchronous iterative Do-All with crashes. *Distributed Computing*. To appear.

[49] Ch. Georgiou, A. Russell, and A.A. Shvartsman. The complexity of synchronous iterative Do-All with crashes. In *Proceedings of the* 15^{th} *International Symposium on Distributed Computing (DISC 2001)*, pages 151–165, 2001.

[50] Ch. Georgiou, A. Russell, and A.A. Shvartsman. The complexity of distributed cooperation in the presence of failures. In *Proceedings of the* 4^{th} *International Conference on Principles of Distributed Systems (OPODIS 2000)*, pages 245–264, 2000.

[51] Ch. Georgiou, A. Russell, and A.A. Shvartsman. Failure-sensitive analysis of parallel algorithms with controlled memory access concurrency. In *Proceedings of the* 6^{th} *International Conference on Principles of Distributed Systems (OPODIS 2002)*, pages 127–138, 2002.

[52] Ch. Georgiou, A. Russell, and A.A. Shvartsman. Work-competitive scheduling for cooperative computing with dynamic groups. In *Proceedings of the* 35^{th} *ACM Symposium on Theory of Computing (STOC 2003)*, pages 251–258, 2003.

[53] Ch. Georgiou and A.A. Shvartsman. Cooperative computing with fragmentable and mergeable groups. *Journal of Discrete Algorithms*, 1(2):211–235, 2003.

[54] Ch. Georgiou and A.A. Shvartsman. Cooperative computing with fragmentable and mergeable groups. In *Proceedings of the* 7^{th} *International Colloquium on Structural Information and Communication Complexity (SIROCCO 2000)*, pages 141–156, 2000.

[55] A. Gharakhani and A.F. Ghoniem. Massively parallel implementation of a 3D vortex-boundary element method. In *Proceedings of the European Series in Applied and Industrial Mathematics*, volume 1, pages 213–223, 1996.

[56] C.G. Gray and D.R. Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *Proceedings of the* 12^{th} *ACM Symposium on Operating Systems Principles (SOSP 1989)*, pages 202–210, 1989.

[57] J.N. Gray. Notes on database operating systems. In R. Bayer, R.M. Graham, and G. Seegmuller, editors, *Operating Systems: An Advanced Course*, volume 60 of *Lecture Notes in Computer Science*, chapter 3.F, pages 393–481. Springer-Verlag, 1978.

[58] S.A. Green. *Parallel Processing for Computer Graphics*. MIT Press/Pitman Publishing, 1991.

[59] 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*, 14(2):75–81, 2001.

[60] V. Hadzilacos and S. Toueg. Fault-tolerant broadcasts and related problems. In *Distributed Systems*, chapter 5, pages 97–145. ACM Press/Addison-Wesley, 1993.

[61] M. Hayden. *The Ensemble System*. PhD thesis, Cornell University, 1998.

[62] M. Hiltunen and R. Schlichting. Properties of membership services. In *Proceedings of the* 2^{nd} *International Symposium on Autonomous Decentralized Systems*, pages 200–207, 1995.

[63] D.R. Hughes and F.C. Piper. *Design Theory*. Cambridge University Press, 1985.

[64] K. Jacobsen, X. Zhang, and K. Marzullo. Group membership and wide-area master-worker computations. In *Proceedings of the* 23^{rd} *IEEE International Conference on Distributed Computing Systems (ICDCS 2003)*, pages 570–581, 2003.

[65] C.B. Jenssen. *Parallel Computational Fluid Dynamics 2000: Trends and Applications*. Elsevier Science Ltd., first edition, 2001.

[66] P.C. Kanellakis, D. Michailidis, and A.A. Shvartsman. Controlling memory access concurrency in efficient fault-tolerant parallel algorithms. *Nordic Journal of Computing*, 2(2):146–180, 1995.

[67] P.C. Kanellakis and A.A. Shvartsman. Efficient parallel algorithms can be made robust. *Distributed Computing*, 5(4):201–217, 1992. A preliminary version appears in the *Proceedings of the* 8^{th} *ACM Symposium on Principles of Distributed Computing (PODC 1989)*, pages 211–222, 1989.

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

[69] R.M. Karp and V. Ramachandran. A survey of parallel algorithms for shared-memory machines. *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity*, pages 869–941, 1990.

[70] Z.M. Kedem, K.V. Palem, M.O. Rabin, and A. Raghunathan. Efficient program transformations for resilient parallel computation via randomization. In *Proceedings of the* 24^{th} *ACM Symposium on Theory of Computing (STOC 1992)*, pages 306–318, 1992.

[71] Z.M. Kedem, K.V. Palem, A. Raghunathan, and P. Spirakis. Combining tentative and definite executions for dependable parallel computing. In *Proceedings of the* 23^{rd} *ACM Symposium on Theory of Computing (STOC 1991)*, pages 381–390, 1991.

[72] Z.M. Kedem, K.V. Palem, and P. Spirakis. Efficient robust parallel computations. In *Proceedings of the* 22^{nd} *ACM Symposium on Theory of Computing (STOC 1990)*, pages 138–148, 1990.

[73] R. Khazan, A. Fekete, and N.A. Lynch. Multicast group communication as a base for a load-balancing replicated data service. In *Proceedings of the* 12^{th} *International Symposium on Distributed Computing (DISC 1998)*, pages 258–272, 1998.

[74] E. Korpela, D. Werthimer, D. Anderson, J. Cobb, and M. Lebofsky. SETI@home: Massively distributed computing for SETI. *Computing in Science and Engineering*, 3(1):78–83, 2001.

[75] D.R. Kowalski and A.A. Shvartsman. Performing work with asynchronous processors: message-delay-sensitive bounds. In *Proceedings of the* 22^{nd} *ACM Symposium on Principles of Distributed Computing (PODC 2003)*, pages 265–274, 2003.

[76] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, 1982.

[77] E. Y. Lotem, I. Keidar, and D. Dolev. Dynamic voting for consistent primary components. In *Proceedings of the* 16^{th} *ACM Symposium on Principles of Distributed Computing (PODC 1997)*, pages 63–71, 1997.

[78] M.C. Loui and H.H. Abu-Amara. Memory requirements for agreement among unreliable asynchronous processes. In F.P. Preparata, editor, *Parallel and Distributed Computing*, volume 4 of *Advances in Computing Research*, pages 163–183. JAI Press, 1987.

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

[80] N.A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, 1996.

[81] N.A. Lynch and M.R. Tuttle. An introduction to Input/Output automata. *CWI Quarterly*, 2(3):219–246, 1989.

[82] G. Malewicz. A work-optimal deterministic algorithm for the asynchronous certified Write-All problem. In *Proceedings of the* 22^{nd} *ACM Symposium on Principles of Distributed Computing (PODC 2003)*, pages 255–264, 2003.

[83] G. Malewicz, A. Russell, and A.A. Shvartsman. Distributed cooperation during the absence of communication. In *Proceedings of the* 14^{th} *International Symposium on Distributed Computing (DISC 2000)*, pages 119–133, 2000.

[84] G. Malewicz, A. Russell, and A.A. Shvartsman. Optimal scheduling for disconnected cooperation. In *Proceedings of the* 8^{th} *International Colloquium on Structural Information and Communication Complexity (SIROCCO 2001)*, pages 259–274, 2001.

[85] C. Martel, A. Park, and R. Subramonian. Work-optimal asynchronous algorithms for shared memory parallel computers. *SIAM Journal on Computing*, 21(6):1070–1099, 1992.

[86] C. Martel and R. Subramonian. On the complexity of certified Write-All algorithms. *Journal of Algorithms*, 16(3):361–387, 1994.

[87] C. Martel, R. Subramonian, and A. Park. Asynchronous PRAMs are (almost) as good as synchronous PRAMs. In *Proceedings of the* 31^{st} *IEEE Symposium on Foundations of Computer Science (FOCS 1990)*, pages 590–599, 1990.

[88] S. Mishra, L.L. Peterson, and R.D. Schlichting. Consul: A communication substrate for fault-tolerant distributed programs. *Distributed Systems Engineering Journal*, 1(2):87–103, 1993.

[89] S. Molnar, J. Eyles, and J. Poulton. PixelFlow: High-speed rendering using image composition. *Computer Graphics*, 26(2):231–240, 1992.

[90] L.E. Moser, Y. Amir, P.M. Melliar-Smith, and D.A. Agarwal. Extended virtual synchrony. In *Proceedings of the* 14^{th} *IEEE International Conference on Distributed Computing Systems (ICDCS 1994)*, pages 56–65, 1994.

[91] L.E. Moser, P.M. Melliar-Smith, D.A. Agarawal, R.K. Budhia, and C.A. Lingley-Papadopolous. Totem: A fault-tolerant multicast group communication system. *Communications of the ACM*, 39(4):54–63, 1996.

[92] Y. Moses and O. Waarts. Coordinated traversal: $(t+1)$-round Byzantine agreement in polynomial time. *Journal of Algorithms*, 17(1):110–156, 1994.

[93] The Olson laboratory fight AIDS@home project. At http://www.fightaidsathome.org.

[94] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, 1980.

[95] D. Powell, editor. *Special Issue on Group Communication Services*, volume 39(4) of *Communications of the ACM*. ACM Press, 1996.

[96] The RSA factoring by web project. At http://www.npac.syr.edu/factoring.

[97] A. Ricciardi, A. Schiper, and K. Birman. Understanding partitions and the "no partition" assumption. In *Proceedings of the* 4^{th} *Workshop on Future Trends of Distributed Computing Systems*, pages 354–360, 1993.

[98] R.L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public key cryptosystems. *Communications of the ACM*, 21:120–126, 1978.

[99] A.L. Rosenberg. Accountable web-computing. In *Proceedings of the* 7^{th} *IEEE International Parallel and Distributed Processing Symposium (IPDPS 2002)*, 2002.

[100] M. Saks, N. Shavit, and H. Woll. Optimal time randomized consensus – making resilient algorithms fast in practice. In *Proceedings of the* 2^{nd} *ACM-SIAM Symposium on Discrete Algorithms (SODA 1991)*, pages 351–362, 1991.

[101] R. Samanta, J. Zheng, T. Funkhouser, K. Li, and J.P. Singh. Load balancing for multi-projector rendering systems. In *SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 107–116, 1999.

[102] R.D. Schlichting and F.B. Schneider. Fail-stop processors: An approach to designing fault-tolerant computing systems. *ACM Transactions on Computing Systems*, 1(3):222–238, 1983.

[103] N. Shavit. *Concurrent Time Stamping*. PhD thesis, The Hebrew University of Jerusalem, 1989.

[104] A.A. Shvartsman. Achieving optimal CRCW PRAM fault-tolerance. *Information Processing Letters*, 39(2):59–66, 1991.

[105] D. Sleator and R. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28(2):202–208, 1985.

[106] D.R. Stinson. *Cryptography: Theory and practice*. CRC PRess, 1995.

[107] J.B. Sussman and K. Marzullo. The bancomat problem: An example of resource allocation in a partitionable asynchronous system. In *Proceedings of the* 12^{th} *International Symposium on Distributed Computing (DISC 1998)*, pages 363–377, 1998.

[108] M. Tambe, J. Adibi, Y. Alonaizon, A. Erdem, G.A. Kaminka, S. Marsella, and I. Muslea. Building agent teams using an explicit teamwork model and learning. *Artificial Intelligence*, 110(2):215–239, 1999.

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

[110] R. van Renesse, K.P. Birman, and S. Maffeis. Horus: A flexible group communication system. *Communications of the ACM*, 39(4):76–83, 1996.

[111] G. Varghese and N. A. Lynch. A tradeoff between safety and liveness for randomized coordinated attack protocols. In *Proceedings of the* 11^{th} *ACM Symposium on Principles of Distributed Computing (PODC 1992)*, pages 241–250, 1992.

[112] S.G. Ziavras and P. Meer. Adaptive multiresolution structures for image processing on parallel computers. *Journal of Parallel and Distributed Computing*, 23(3):475–483, 1994.