Performing Tasks on Restartable Message-Passing Processors*

Bogdan S. Chlebus[1] and Roberto De Prisco[2] and Alex A. Shvartsman[3]

[1] Instytut Informatyki, Uniwersytet Warszawski,
Banacha 2, 02-097 Warszawa, Poland.
chlebus@mimuw.edu.pl
[2] Laboratory for Computer Science, Massachusetts Institute of Technology,
545 Technology Square NE43-368, Cambridge, MA 02139, USA.
robdep@theory.lcs.mit.edu
[3] Department of Computer Science and Engineering, University of Connecticut,
191 Auditorium Road, U-155, Storrs, CT 06269, USA.
aas@eng2.uconn.edu

Abstract. This work presents new algorithms for the "Do-All" problem that consists of performing t tasks reliably in a message-passing synchronous system of p fault-prone processors. The algorithms are based on an aggressive coordination paradigm in which multiple coordinators may be active as the result of failures. The first algorithm is tolerant of $f < p$ stop-failures and it does not allow restarts. It has the available processor steps complexity $S = O((t + p \log p / \log\log p) \cdot \log f)$ and the message complexity $M = O(t + p \log p / \log\log p + f \cdot p)$. Unlike prior solutions, our algorithm uses redundant broadcasts when encountering failures and, for large f, it has better S complexity. This algorithm is used as the basis for another algorithm which tolerates any pattern of stop-failures *and restarts*. This new algorithm is the first solution for the Do-All problem that efficiently deals with processor restarts. Its available processor steps complexity is $S = O((t + p \log p + f) \cdot \min\{\log p, \log f\})$, and its message complexity is $M = O(t + p \cdot \log p + f \cdot p)$, where f is the number of failures.

1 Introduction

The problem of performing t tasks reliably and in parallel using p processors is one of the fundamental problems in distributed computation. This problem, which we call Do-All, was considered for the synchronous message-passing model

* This work was supported by the following contracts: ARPA N00014-92-J-4033 and F19628-95-C-0118, NSF 922124-CCR, ONR-AFOSR F49620-94-1-01997, and DFG-Graduiertenkolleg "Parallele Rechnernetzwerke in der Produktionstechnik" ME 872/4-1, DFG-SFB 376 "Massive Parallelität: Algorithmen, Entwurfsmethoden, Anwendungen". The research of the third author was substantially done at the Massachusetts Institute of Technology. The research of the first and the third authors was partly done while visiting Heinz Nixdorf Institut, Universität-GH Paderborn.

by Dwork, Halpern and Waarts in their pioneering work [2]. They developed several efficient algorithms for this problem in the setting where the processors are subject to fail-stop (or crash) failures and where the tasks can be performed using the *at-least-once* execution semantics (i.e., the tasks either are or can be made idempotent). In the setting of [2], the cost of local computation, whether performing low-level administrative tasks or idling, is considered to be negligible compared to the costs of performing each of the t tasks.

In solving Do-All, Dwork, Halpern and Waarts define the *effort* of an algorithm as the sum of the work complexity (i.e., the number of tasks executed, counting multiplicities) and message complexity (i.e., the number of messages used). This approach to efficiency does not account for any steps spent by processors waiting for messages or time-outs. This allows algorithm optimizations which keep the number of messages small, because processors can afford to wait to obtain sufficient information by not receiving messages in specific time intervals.

De Prisco, Mayer and Yung also consider the Do-All problem without processor restarts in their study [1]. Their goal is the development of fast and message-efficient algorithms. The work measure they consider is the available processor steps S (introduced by Kanellakis and Shvartsman [6]). This measure accounts for *all* steps taken by the processors, that is, the steps involved in performing the Do-All tasks and any other computation steps taken by the processors. Optimization of S leads to fast algorithms whose performance degrades gracefully with failures. The communication efficiency is gauged using the standard message complexity measure. The authors successfully pursue algorithmic efficiency in terms of what they call the *lexicographic optimization* of complexity measures. This means firstly achieving efficient work, then efficient communication complexity.

A similar approach to efficiency is pursued by Galil, Mayer and Yung [3] who also derive a very efficient Do-All solution for stop-failures.

Our contributions. In this paper we solve the Do-All problem in the setting where the p processors are subject to dynamic stop-failures *and restarts*. The complexity concerns in this paper follow the criteria established in [1]. We seek algorithmic efficiency with respect to both the work, expressed as available processor steps S, and the communication, expressed as the message complexity M. We want to minimize S, having M as small as possible.

We introduce an aggressive coordinator scheduling paradigm that allows multiple coordinators to be active concurrently. Because multiple coordinators are activated only in response to failures, our algorithms achieve efficiency in S and M.

It is not difficult to formulate trivial solutions to Do-All in which each processor performs each of the t tasks. Such solutions have work $\Omega(t \cdot (p+r))$, where r is the number of restarts, and they do not require any communication. Thus work-efficient solutions need to trade messages for work. Our solution is the first non-trivial efficient algorithm tolerant of stop-failures and restarts determined by the the worst-case omniscient adversary.

En route to the solution for restartable processors we introduce a new algorithm for the Do-All problem without restarts. This algorithm, that we call "algorithm AN" (Algorithm No-restart), is tolerant of $f < p$ stop-failures. It has available processor steps complexity[4] $S = O((t + p\log p/\log\log p) \cdot \log f)$ and message complexity $M = O(t + p\log p/\log\log p + f \cdot p)$.

Algorithm AN is the basis for our second algorithm, called "algorithm AR" (Algorithm with Restarts), which tolerates any number of stop-failures and restarts. Algorithm AR is the *first* such solution for the Do-All problem. Its available processor steps complexity is $S = O((t+p\log p+f)\cdot\min\{\log p, \log f\})$, and its message complexity is $M = O(t+p\cdot\log p+f\cdot p)$, where f is the number of failures.

Our algorithm AN is more efficient in terms of S than the algorithms of [1] and [3] when f, p and t are comparable; the algorithm also has efficient message complexity. Both algorithm AN and algorithm AR come within a $\log f$ (and $\log p$) factor of the lower bounds [6] for any algorithms that balance loads of surviving processors in each constant-time step. We achieve this by deploying an aggressive processor coordination strategy, in which more than one processor may assume the role of the *coordinator*, the processor whose responsibility is to ensure the progress of the computation. This approach is suggested by the observation that algorithms with only one coordinator cannot efficiently cope with restarts. Indeed the real advantage of this approach is that it can be naturally extended to deal with processor failures and restarts, with graceful deterioration of performance.

The improvements in S, however, come at a cost. Both of our algorithms assume reliable multicast [4]. Prior solutions do not make this assumption, although they do not solve the problem of processor restarts. The availability of reliable broadcast simplifies solutions for non-restartable processors, but dealing with processor restarts remains a challenge even when such broadcast is available. There are several reasons for considering solutions with reliable multicasts. First of all, in a distributed setting where processors cooperate closely, it becomes increasingly important to assume the ability to perform efficient and reliable broadcast or multicast. This assumption might not hold for extant WANs, but it is true for broadcast LANs (e.g., Ethernet and bypass rings). The availability of hardware-assisted broadcast makes the cost of using the broadcast communication comparable to the cost of sending a single point-to-point message. Note however that we are using a conservative cost measure which assumes that the cost of a multicast is proportional to the number of recipients. Secondly, by separating the concerns between the reliability of processors and the underlying communication medium, we are able to formulate solutions at a higher level of modularity so that one can take advantage of efficient reliable broadcast algorithms (cf. [4]) without altering the overall algorithmic approach. Lastly, our approach presents a new venue for optimizing Do-All solutions and for beating the $\Omega(t + (f + 1) \cdot p)$ lower bound of stage-checkpointing algorithms [1].

[4] All logarithms are to the base 2; the expression "$\log f$" stands for 1 when $f < 2$ and $\log_2 f$ otherwise.

Review of prior work. Dwork, Halpern and Waarts [2] developed the first algorithms for the Do-All problem. One algorithm presented by the authors (protocol $\mathcal{B}$) has effort $O(t + p\sqrt{p})$, with work contributing the cost $O(t + p)$ towards the effort, and message complexity contributing the cost $O(p\sqrt{p})$. The running time of the algorithm is $O(t+p)$. Another algorithm in [2] (protocol $\mathcal{C}$) has effort $O(t + p\log p)$. This includes optimal work of $O(t + p)$, message complexity of $O(p\log p)$, and time $O(p^2(t + p)2^{t+p})$. Thus the reduction in message complexity is traded-off for a significant increase in time. The third algorithm (protocol $\mathcal{D}$) obtains work optimality and is designed for maximum speed-up, which is achieved with a more aggressive checkpointing strategy, thus trading-off time for messages. The message complexity is quadratic in p for the fault-free case, and in the presence of a failure pattern of $f < p$ failures, the message complexity degrades to $\Theta(f \cdot p^2)$.

De Prisco, Mayer and Yung [1] present an algorithm which has the available processor steps $O(t + (f + 1)p)$ and message complexity $O((f + 1)p)$. The available processor steps and communication efficiency approach requires keeping all the processors busy doing tasks, simultaneously controlling the amount of communication. De Prisco, Mayer and Yung were the first to report results on Do-All algorithms in the fail-stop case using this efficiency approach. To avoid the quadratic upper bound for S substantial processing slackness ($p \ll t$) is assumed. In [1] a lower bound of $\Omega(t + (f + 1)p)$ for algorithms that use the stage-checkpointing strategy is proved. However there are algorithmic strategies that have the potential of circumventing the quadratic bound. Consider the following scenarios. In the first scenario we have $t = o(p)$, $f > p/2$, and the algorithm assigns all tasks to every processor. Then $S = O(p \cdot t) = o(t + (f + 1) \cdot p)$, because $f \cdot p = \Theta(p^2)$. This naïve algorithm has a quadratic work performance for $p = O(t)$. In the second example assume that the three quantities p, t and f are of comparable magnitude. Consider the algorithm in which all the processors are coordinators, work is interleaved with communication, and the outstanding work is evenly allocated among the live processors based on their identifiers. The work allocation is done after each round of exchanging messages about which processors are still available and which tasks have been successfully performed. One can show that $S = O(p \cdot \log p/ \log\log p)$. This bound is $o(t + (f + 1) \cdot p)$ for $f > p/2$ and $t = p$. Unfortunately the number of messages exchanged is more than quadratic, and can be $\Omega(p^2 \cdot \log p/ \log\log p)$. These examples suggest a possibility of improvement of the bound $S = O(t + (f + 1)p)$, however the simple algorithms discussed above have either the available processor steps quadratic in p, or the number of messages more than quadratic in p in the case when p, t and f are of the same order. One interesting result of our paper is showing that an algorithm can be developed which has both the available processor steps which is always subquadratic, and the number of messages which is quadratic only for f comparable to p, even with restarts.

The algorithm in [1] is designed so that at each step there is at most one coordinator; if the current coordinator fails then the next available processor takes over, according to a time-out strategy. Having a single coordinator helps to bound the number of messages, but a drawback of such approach is that any protocol with at most one active coordinator is bound to have $S = \Omega(t+(f+1)\cdot p)$. Namely, consider the following behavior of the adversary: each coordinator is stopped immediately after it becomes one and before it sends any messages. This creates pauses of at least $O(1)$ steps, giving the $\Omega((f+1)\cdot p)$ part. Eventually there remains only one processor which has to perform all the tasks, because it has never received any messages, this gives the remaining $\Omega(t)$ part. A related lower-bound argument for stage-checkpointing strategies is formally presented in [1]. Moreover, when processor restarts allowed, any algorithm that relies on a single coordinator for information gathering might not terminate (the adversary can always kill the current coordinator, keeping alive all the other processors so that no progress is made).

Another important algorithm was developed by Galil, Mayer and Yung [3]. Working in the context of Byzantine agreement with stop-failures (for which they establish a message-optimal solution), they improved the message complexity of [1] to $O(f \cdot p^{\varepsilon} + \min\{f+1, \log p\}p)$, for any positive ε, while achieving the available processor steps complexity of $O(t+(f+1)\cdot p)$.

The Do-All problem for the shared-memory model of computation, where it is called *Write-All*, was introduced and studied by Kanellakis and Shvartsman [6, 7]. Parallel computation using the iterated Do-All paradigm is the subject of several subsequent papers, most notably the work of Kedem, Palem and Spirakis [8], Martel, Park and Subramonian [11] and Kedem, Palem, Rabin and Raghunathan [9].

Kanellakis, Michailidis and Shvartsman [5] developed a technique for controlling redundant concurrent access to shared memory in algorithms with processor stop-failures. This is done with the help of a structure they call *processor priority tree*. In this work we use a similar structure in the qualitatively different message-passing setting. Furthermore, we are able to use our structure with restartable processors.

The structure of the rest of the paper is as follows. Section 2 contains definitions and gives a high-level view of the algorithms. Section 3 includes the presentation of algorithm AN with a proof of its correctness and analysis. Section 4 gives algorithm AR with correctness and analysis. The final Section 5 concludes with remarks and future work. The optional appendix contains proof sketches.

2 Definitions and algorithmic preliminaries

In this section we describe the model of distributed computation, the failure models, and we introduce the main ideas underlying our algorithms.

2.1 Model

We consider a distributed system consisting of a set $\mathcal{P}$ of p processors. Processors communicate only by message passing at the level of abstraction of the *network layer*, i.e., any processor can send messages to any other processor and the contents of messages are not corrupted. We assume that the set $\mathcal{P}$ is fixed and is known to all processors in $\mathcal{P}$. Processors have unique identifiers (PIDs) and the set of PIDs is totally ordered. The distributed system is synchronous and we assume that there is a global clock available to all the processors. Between each two consecutive clock ticks a processor takes a *step* during which the processor can receive messages, perform some local computation and send messages. For the sake of clarity of presentation we think of a step as further subdivided into three substeps: during the first one a processor receives messages sent to it during the previous step, during the second substep a processor performs some local computation, and during the third substep a processor may send some messages. We refer to these substeps as the *receive* substep, the *compute* substep and the *send* substep.

We define a *task* to be a computation that can be performed by any processor in unit time. Tasks are uniquely identified by their UIDs and the set of UIDs is totally ordered. Our distributed system has to perform t tasks with UIDs in the set $\mathcal{T}$ ($t = |\mathcal{T}|$). The tasks are *idempotent*, i.e., each can be performed using the *at-least-once* execution semantics. Initially, the set $\mathcal{T}$ of tasks is known to all the processors. A task can be performed during the compute substep together with some local computation.

We consider two processor failure models: the *fail-stop* model in which processors do not restart after a failure, and the *fail-stop/restart* model in which restarts are allowed. In either model any processors may stop at any moment during the computation. Such a processor does not receive any messages and does not perform any computation. In the fail-stop/restart model, a processor can restart at any point after a failure. Upon a restart the state of the restarted processor is reset to an initial state, but the processor is aware of the restart. Any messages sent to a processor prior to its restart are lost. We assume that during a single step a stopped processor can restart at most once (e.g., a processor can restart in response to a clock tick).

We define an execution to be a sequence of *steps* during which some number of processors, in parallel, perform their send, compute and send substeps. Given a particular finite execution we denote by f the number of actual failures and by r the number of actual restarts. For the fail-stop model we assume that at least one processor operational at any time, i.e., for any finite prefix of any execution we have $r = 0$ and that $f < p$. In the fail-stop/restart model it is possible to relax the assumption that there exists an infallible processor. The natural generalization of the condition $f < p$ is: for any finite prefix of any execution we have $f < r+p$, i.e., during each step there is at least one operational processor. However this condition turns out to be too weak because it allows for all information about progress to be lost. For example, consider the scenario in which half of the processors are alive initially, they perform some tasks, and then they all crash while the other half restarts. This can be repeated forever without any globally known progress. Thus we require a stronger condition which assumes that for any two consecutive "phases", where a phase is some small constant number of consecutive steps specific to an algorithm, there is at least one processor that is operational through the two phases. This condition rules out *thrashing* adversaries that repeatedly stop and restart processors in such a way that any progress made by the computation is lost (like in the above example).

We assume that reliable multicast [4] is available. With reliable multicast a processor q can send a message to any set $P \subseteq \mathcal{P}$ of processors in its send substep. All processors in P that are operational during the entire following receive substep receive the message sent by q.

Our goal is to execute the tasks in $\mathcal{T}$ efficiently, where the efficiency is measured in terms of the *available processor steps* S and the *communication complexity*. The available processor steps S is defined by the stipulation that any processor being operational during a time step contributes a unit to S. Formally, if p_i is the number of processors operational during step i then $S = \sum_{i=1}^{\delta} p_i$, where δ is the last step of the computation. The communication complexity M is the number of point-to-point messages sent by processors. Each message sent from a processor q_1 to processor q_2 (whether faulty or not) contributes a unit to M. During each step a processor can send at most one message to any of the other $p-1$ processors. We are not concerned with the size of messages; however, using bit-string set encoding, each message sent contains $O(\max\{t,p\})$ bits.

2.2 Overview of algorithmic techniques

Computation proceeds in a loop, which is repeated until all the tasks are done. An iteration of the loop is referred to as a *phase*. A phase consists of some constant number of consecutive steps (we use three steps for each phase). Because any phase consists of a constant number of steps, the available processor steps is $S = O(\sum_{\ell} p_{\ell})$, where p_{ℓ} is the number of processors taking at least one step in phase ℓ and the sum is over all phases of the execution of the algorithm.

Since we consider stop-failures, a processor can be in one of the following two states: *live*, when it is operational, or *stopped*, otherwise. For a given execution, the number f (resp. r) of failures (resp. restarts) is defined as the number of processor state changes from live to stopped (resp. from stopped to live). These state changes may occur at any point in the course of a phase. Throughout the rest of the paper we use the following terminology.

Definition 1. A processor is said to be:

- "available in phase ℓ", if it is alive at the beginning of the phase;
- "active in phase ℓ", if it is available in phase ℓ and sends all the messages it is supposed to send in phase ℓ;
- "restarted in phase ℓ" if it is not available in phase $\ell-1$ but it is available in phase ℓ;
- "failed in phase ℓ" if it is available in phase ℓ but it is not available in phase $\ell+1$.

This definition does not take into account the cases where a processor restarts and then fails shortly after the restart, without becoming available for the subsequent phase. We refer to such restarts as *false restarts*.

A processor can be a *coordinator* of a given phase. All available processor (including coordinators) are also *workers* in a given phase. Coordinators are responsible for recording progress, while workers respond to coordinators' inquiries and perform tasks in response to coordinators' requests. There may be multiple coordinators in a given phase.

Coordinator appointments. The number of processors which 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. This guarantees that there can be $O(\log p)$ consecutive phases without active coordinators unless all processors stop. There are $\Theta(\log p)$ such phases only if the number of failures is $\Omega(p)$. Whenever at least one coordinator is active in a phase, the number of coordinators for the next phase is reduced to one. Allowing an exponential rate of growth in the number of coordinators seems to be an expensive strategy but we show that it is viable and efficient.

Local views. Processors assume the coordinator role based on their local knowledge. During the computation each processor w maintains a sequence $L_w = \langle q_1, q_2, ..., q_k\rangle$ of PIDs of potentially available processors. We call such list a *local view*, and we let $P_w = \{q_1, q_2, ..., q_k\}$ to be the set of PIDs in L_w. The PIDs in L_w are partitioned into *layers* consisting of consecutive PIDs: $L_w = \langle q_1, q_2, ..., q_k\rangle = \langle \Lambda^0, \Lambda^1, \Lambda^2, ..., \Lambda^{j_k}\rangle$[5]. When $\Lambda^0 = \langle q_1\rangle$ the layered structure can be visualized in terms of a complete binary tree rooted at processor q_1, where nodes are placed from left to right with respect to the linear order given by L_w; thus, in a tree-like layered structure, layer Λ^0 consists of processor q_1, layer Λ^i consists of 2^i consecutive processors starting at processor q_{2^i} and ending at processor $q_{2^{i+1}-1}$ (see Figure 1).

Layer Λ^0	5
Layer Λ^1	17 12
Layer Λ^2	14 1 16 7
Layer Λ^3	15 9 10 11 3 13 4 8
Layer Λ^4	6 2

Fig. 1. An example showing the layered structure with processors $\langle 5, 17, 12, 14, 1, 16, 7, 15, 9, 10, 11, 3, 13, 4, 8, 6, 2\rangle$.

The local view is used to implement the martingale principle of appointing coordinators as follows. Let $L_{\ell,w} = \langle \Lambda^0, \Lambda^1, \Lambda^2, ..., \Lambda^{j_k}\rangle$ be the local view of worker w at the beginning of phase ℓ. Then processor w expects processors in layer Λ^0 to act as coordinators in phase ℓ; in the case layer Λ^0 is not active in phase ℓ, then processor w expects layer Λ^1 to be active in phase $\ell + 1$; in general processor w expects layer Λ^i to be active in phase $\ell + i$ if all previous layers Λ^j, $\ell \leq j < \ell + i$, were not active in phase $\ell + j$. The local view is updated at the end of each phase.

[5] For sequences $L = \langle e_1, \ldots, e_n\rangle$ and $K = \langle d_1, \ldots, d_m\rangle$ we define $\langle L, K\rangle$ to be the sequence $\langle e_1, \ldots, e_n, d_1, \ldots, d_m\rangle$.

Example. Let the local view of a worker w for phase ℓ be the one in Figure 1. Then a possible view for processor w for phase $\ell + 2$ is the one in Figure 2. Processor w view may get to this view in phase $\ell + 2$, if processor 5 is not active in phase ℓ and processors 17, 12 are not active in phase $\ell + 1$. Subsequently, the local view of processor w can be the one in Figure 3. Processor w may get to this view in phase $\ell + 4$ if, for example, processors 14, 1, 16, 7 are not active in phase $\ell + 2$ and in phase $\ell + 3$ processors 15, 9, 11, 3, 13, 4 are active, processors 8 and 10 are failed and processors 1 and 16 are restarted.

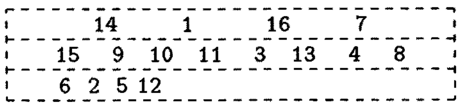

Fig. 2. The local view for phase $\ell + 2$.

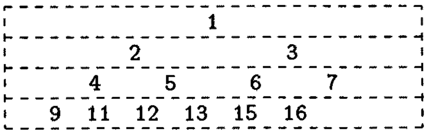

Fig. 3. The local view for phase $\ell + 4$.

Allocating tasks and the load balancing rule. During the execution each processor w keeps its local information about the set D_w of units of tasks already performed, and the set P_w of live processors. Set D_w is always an underestimate of the set of tasks actually done and P_w is always an overestimate of the set of processors that are available. We denote by U_w the set of *unaccounted* tasks, i.e., whose done status is unknown to w. Sets U_w and D_w are related by $D_w = \mathcal{T} \setminus U_w$, where $\mathcal{T}$ is the set of all the tasks. Given a phase ℓ we use $P_{\ell,w}$, $U_{\ell,w}$ and $D_{\ell,w}$ to denote the values of the corresponding sets at the beginning of phase ℓ. Consider a phase ℓ and let w be a worker active in phase ℓ. Let i be the rank of processor w in the layered structure $L_{\ell,w}$. The *load balancing rule* tells worker w to execute the $(i \bmod |U_{\ell,w}|)^{th}$ unit of work.

Algorithm structure. At the beginning of phase ℓ processor w knows the local view $L_{\ell,w}$ (and thus the set $P_{\ell,w}$) and the set $U_{\ell,w}$ of unaccounted tasks (and thus the set $D_{\ell,w}$ of accounted tasks). Each processor performs one task according to the load balancing rule and attempts to report the execution of the task to any coordinator of phase ℓ. Any live coordinator c gathers reports from the workers, updates its information about $P_{\ell,c}$ and $U_{\ell,c}$ and broadcasts this new information causing local views to be reorganized. We will see that at the beginning of any phase ℓ all live processors have the same local view L_ℓ and the same set U_ℓ of unaccounted tasks and that accounted tasks have been actually executed. A new phase starts if U_ℓ is not empty.

3 No restarts – algorithm AN

In this section we define algorithm AN for the fail-stop model. Although solving Do-All using the machinery we assume is relatively easy, we develop algorithm AN as the basis for algorithm AR which solves the Do-All problem in the more general fail-stop/restart model.

Structure of a phase. A phase consists of 3 steps.

S1. The receive substep is not used. In the compute substep, any worker w performs a specific task u according to the load balancing rule. In the send substep the worker w sends a `report`(u) to any known coordinator.

S2. In the receive substep the coordinators gather `report` messages. For any coordinator c, let $u_c^1, ..., u_c^{k_c}$ be the set of task UIDs received. In the compute substep c sets $D_c \leftarrow D_c \cup \bigcup_{i=1}^{k_c} \{u_c^i\}$, and P_c to the set of worker PIDs from which c received `report` messages. In the send substep, coordinator c multicasts the message `summary`(D_c, P_c) to processors in P_c.

S3. During the receive substep `summary` messages are received by live processors. For any worker w, let $(D_w^1, P_w^1), ..., (D_w^{k_w}, P_w^{k_w})$ be the sets received in `summary` messages. In the compute step w sets $D_w \leftarrow D_w^i$ and $P_w \leftarrow P_w^i$ for an arbitrary $i \in \{1, ..., k_w\}$. The worker w also updates its local view L_w as described below. The send substep is not used.

Updating the local view. Initially (phase 0) the local view $L_{0,w}$ of any processor w is defined as the set of processors $\mathcal{P}$ structured in layers as a tree-like layered structure given in Section 2. Let us consider a generic phase ℓ and let the local view of processor w for phase ℓ be $L_{\ell,w} = \langle q_1, q_2, ..., q_k \rangle = \langle \Lambda^0, \Lambda^1, ..., \Lambda^{j_k} \rangle$. We distinguish two possible cases.

CASE 1. No coordinators are active in phase ℓ. Then the local view of processor w for phase $\ell + 1$ is $L_{\ell+1,w} = \langle \Lambda^1, ..., \Lambda^{j_k} \rangle$.

CASE 2. When at least one coordinator is active in phase ℓ, processor w receives messages from some coordinator in Λ^0. Processor w computes its set P_w as described in step S3 (we will see that all workers compute the same set P_w). The local view $L_{\ell+1,w}$ of w for phase $\ell + 1$ is the tree-like structure with processors in P_w ordered by their PIDs.

A generic phase is depicted in Figure 4 in Section 4 (for algorithm AN ignore the messages and steps of the restarted processors).

Correctness and efficiency. We first prove that algorithm AN correctly solves the Do-All problem. We start by showing that at the beginning of each phase every available processor has consistent knowledge of the ongoing computation. Then we prove safety (no live processor or undone task is forgotten) and progress properties (tasks execution) which imply the correctness of the algorithm.

Lemma 2 (AN:Consistency). *In any execution of algorithm* AN, *for any two processors w, v available in phase ℓ, we have that $L_{\ell+1,w} = L_{\ell+1,v}$ and that $U_{\ell+1,w} = U_{\ell+1,v}$.*

Because of the previous lemma, we can define $L_\ell = L_{\ell,w}$ for any w as the view at the beginning of phase ℓ, $P_\ell = P_{\ell,w}$ as the set of available processors, $D_\ell = D_{\ell,w}$ as the set of done tasks and $U_\ell = U_{\ell,w}$ as the set of unaccounted tasks at the beginning of phase ℓ.

Lemma 3 (AN:Safety1). *In any execution of algorithm* AN, *if a processor w is active in phase $\ell - 1$ then processor w belongs to P_ℓ.*

Lemma 4 (AN:Safety2). *In any execution of algorithm* AN, *if a task u has not been executed in phases $1, 2, ..., \ell - 1$ then u belongs to U_ℓ.*

We say that a phase ℓ is *attended* if at least one of the processor supposed to be coordinator according to the view L_ℓ is active during phase ℓ. Otherwise the phase is *unattended*.

Let us denote the set of all the attended phases by $A = \{\alpha_1, \alpha_2, ..., \alpha_\tau\}$, for $\alpha_1 < \alpha_2 < ... < \alpha_\tau$ and a given particular execution of algorithm AN. Let us denote by π_i the unattended phases in between the attended phases α_i and α_{i+1}. We refer to π_i as the i^{th} (unattended) period; an unattended period can be empty. Hence the computation proceeds as follows: unattended period π_0, attended phase α_1, unattended period π_1, attended phase α_2, and so on. After the last attended phase α_τ, the algorithm terminates. Indeed since there are no other attended iterations it must be the case that there are no tasks left unaccounted after phase α_τ. We denote by p_i the cardinality of the set of available processors for phase i, i.e., $p_i = |P_i|$, and by u_i the cardinality of the set of unaccounted tasks for phase i, i.e., $u_i = |U_i|$. We let $u_1 = t$ and $u_{\tau+1} = 0$.

Lemma 5 (AN:Progress1). *In any execution of algorithm* AN, *for any attended phase ℓ we have that $u_\ell > u_{\ell+1}$.*

Lemma 6 (AN:Progress2). *In any execution of algorithm* AN, *any unattended period consists of at most* $\log f$ *phases.*

Theorem 7 (AN:Correctness). *In any execution of algorithm* AN *such that $f < p$, i.e., at least one processor survives, the algorithm terminates and all the units of work are performed.*

To assess 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 S is $S_a + S_u$.

The following lemma uses the construction by Martel [10, 6].

Lemma 8. *In any execution of algorithm* AN, $S_a = O(t + p\log p/\log\log p)$.

Lemma 9. *In any execution of algorithm* AN, $S_u = O(S_a \cdot \log f)$.

Theorem 10. *In any execution of algorithm* AN, *the available processor steps is* $S = O(\log f \cdot (t + p\log p/\log\log p))$.

Thus the work of algorithm AN is within a $\log f$ (and hence also $\log p$) factor of the lower bound of $\Omega(t + p\log p/\log\log p)$ [6] for any algorithm that performs tasks by balancing loads of surviving processors in each time step.

For each attended phase $\alpha_i \in A$, let d_i be some distinguished active coordinator, we refer to d_i as the *designated coordinator* of phase α_i. Let M_{d_i} be the number of messages sent or received in phase α_i by d_i. We denote by $M_d = \sum_{i=1}^{\tau} M_{d_i}$ the number of messages sent and received by the designated coordinators during all the attended phases. Let M_f be the number of all other messages, i.e., both the messages sent in unattended periods and the messages sent and received in attended phases by the non-designated coordinators.

Lemma 11. *In any execution of algorithm* AN, $M_d = O(S_a)$.

Lemma 12. *In any execution of algorithm* AN, $M_f = O(f \cdot p)$.

Theorem 13. *In any execution of algorithm* AN, *the number of messages sent is* $M = O(t + p\log p/\log\log p + f \cdot p)$.

4 Stop-failures and restarts – algorithm AR

In this section we describe algorithm AR which solves Do-All in the model of stop failures with restarts. This algorithm is obtained by modifying algorithm AN. The condition that the number of failures is $f < r+p$ provides the condition analogous to $f < p$ of the fail-stop model.

Algorithm AR is similar to algorithm AN; the difference is that there are added messages to handle the restart of processors. A stopped processor q may become live at any moment. At the moment of the restart, processor q has the initial information about the set $\mathcal{P}$ of processors and the set $\mathcal{T}$ of tasks but no information about the ongoing computation.

The steps S1, S2 and S3 in the phase in algorithm AR are similar to those of algorithm AN. After the restart, processor q broadcasts `restart`(q) messages in the send substep of each step until it receives a response. Processors receiving such messages, ignore them if these messages are not received by a certain point within a phase. Thus we can imagine that a restarted processor q broadcasts a `restart`(q) in step S1 of a phase ℓ. This message is then received by all the live and restarted processors of that phase, and, as we will see shortly, processor q is re-integrated in the view for the phase $\ell + 1$. Moreover processor q needs to be informed about the status of the ongoing computation. Hence all the processors who have been live since the start of S1 send an `info`(U_ℓ, L_ℓ) to such q with the set U_ℓ of unaccounted tasks and the local view L_ℓ.

Structure of a phase ℓ. (See Figure 4.)

S1. The receive substep is not used. In the compute substep any worker w performs a specific task u according to the load balancing rule. In the send substep w sends a `report`(u) to any known coordinator. Any restarted processor q broadcasts the `restart`(q) message informing all live processors of its restart.

S2. In the receive step the coordinators gather `report` messages and all live processors gather `restart` messages. Let R be the set of processors that sent a `restart` message. For any coordinator c, let $u_c^1, ..., u_c^k$ be the set of task UIDs received. In the compute substep c sets $D_c \leftarrow D_c \cup \bigcup_{i=1}^{k_c} \{u_c^i\}$ and P_c to be the set of workers from which c received `report` messages. In the send substep, coordinator c multicasts the message `summary`(D_c, P_c) to the available and restarted processors. Any available processor also sends the message `info`(U_ℓ, L_ℓ) to processors in R.

S3. Restarted processors in R receive `info`(U_ℓ, L_ℓ) messages. A restarted processor q sets $L_q \leftarrow L_\ell$ and $U_q \leftarrow U_\ell$. Let $(D_w^1, P_w^1), ..., (D_w^{k_w}, P_w^{k_w})$ be the sets received in `summary` messages by processor w which received such messages. Processor w sets $D_w \leftarrow D_w^i$ and $P_w \leftarrow P_w^i$ for an arbitrary $i \in 1, ..., k$ and $U_w \leftarrow \mathcal{T} \setminus D_w$. Each processor w updates its local view L_w as described below. The send substep is not used.

Layered structure reorganization. Initially (phase 0) the local view $L_{0,w}$ of any processor w is defined as the set of processors $\mathcal{P}$ structured in layers as a tree-like layered structure given in Section 2. Let us consider a generic phase

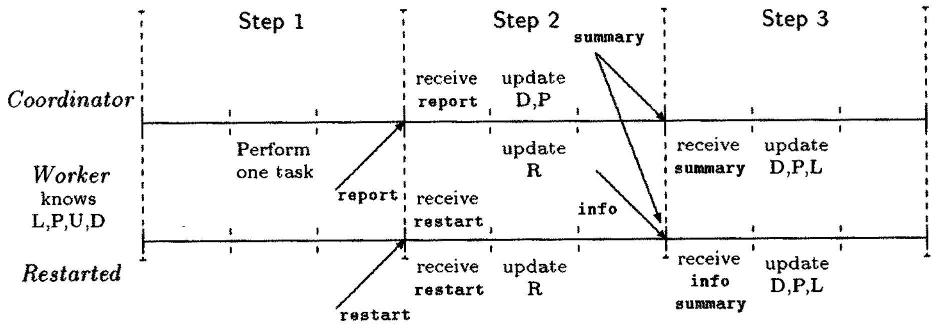

Fig. 4. A phase of algorithm AR (for algorithm AN ignore restarts).

ℓ and let the local view of processor w for phase ℓ be $L_{\ell,w} = \langle q_1, q_2, ..., q_k \rangle = \langle A^0, A^1, ..., A^{j_k} \rangle$. We distinguish three possible cases.

Case 1. In phase ℓ no coordinator is active and no processor restarts. Then the algorithm proceeds exactly as in the no restart case: the local view of processor w for phase $\ell + 1$ is $L_{\ell+1,w} = \langle A^1, ..., A^{j_k} \rangle$.

Case 2. In phase ℓ no coordinator is active but some processors restart. Let R^ℓ be the set of restarted processors who succeed in sending the `restart` messages. Let R' be the set of processors of R^ℓ that are not already in the local view $L_{\ell,w}$. Let $\langle R' \rangle$ be the processors in R' ordered according to their PIDs. The local view for the next phase is $L_{\ell+1,w} = \langle A^1, ..., A^{j_k} \rangle \oplus \langle R' \rangle$. The operator $\oplus$ places processors of R', in the order $\langle R' \rangle$, into the last layer A^{j_k} till this layer contains exactly the double of the processors of layer $A^{j_k - 1}$ and possibly adds a new layer $A^{j_k + 1}$ to accommodate the remaining processors of $\langle R' \rangle$. That is, newly restarted processors which are not yet in the view, are appended at the end of the old layered structure. Notice that restarted processors which receive `info` messages know the old view L_ℓ.

Case 3. In phase ℓ there are both active coordinators and restarted processors. Since there are active coordinators, `summary` messages are received by available, live and restarted processors. Processor w sets P_w as described in step 3; moreover processor w knows the set R'. The new layered structure $L_{\ell+1,w}$ for the next phase consists of all the processors in $P_w \cup R'$, ordered according to their PIDs and the layered structure is the tree-like layered structure.

Correctness and efficiency. The proof of correctness is similar to that used for algorithm AN. The definitions of terms and of S_a, S_u, M_d and M_c carry over.

Lemma 14 (AR:Consistency). *In any execution of algorithm* AR, *for any two processors w, v available in phase ℓ, we have that $L_{\ell+1,w} = L_{\ell+1,v}$ and that $U_{\ell+1,w} = U_{\ell+1,v}$.*

Lemma 15 (AR:Safety1). *In any execution of algorithm* AR, *if a processor w is active or restarted in phase $\ell - 1$, then processor w belongs to P_ℓ.*

Lemma 16 (AR:Safety2). *In any execution of algorithm* AR, *if a task u has not been executed in phases $1, 2, ..., \ell - 1$ then u belongs to U_ℓ.*

Lemma 17 (AR:Progress1). *In any execution of algorithm* AR, *for any attended phase ℓ we have that $u_\ell > u_{\ell+1}$.*

Lemma 18 (AR:Progress2). *In any execution of algorithm* AR, *any unattended period consists of at most* $\min\{\log p, \log f\}$ *phases.*

Theorem 19 (AR:Correctness). *In any execution of algorithm* AR *such that $f < r + p$ with at least one processor active in any two consecutive phases the algorithm terminates and all the units of work are performed.*

We next analyze the performance of algorithm AR in terms of the available processor steps S used and the number M of messages sent.

Lemma 20. *In any execution of algorithm* AR, $S_a = O(t + p\log p + f)$.

Lemma 21. *In any execution of algorithm* AR, $S_u = O(S_a + f) \cdot \min\{\log p, \log f\})$

Theorem 22. *For any execution of algorithm* AR, $S = O((t + p\log p + f) \cdot \min\{\log p, \log f\})$.

For each attended phase $\alpha_i \in A$, let d_i (designated coordinator) be some specific active coordinator, and M_{d_i} denote the number of messages sent or received in phase α_i by d_i, with the exception of the `restart` messages. $M_d = \sum_{i=1}^{\tau} M_{d_i}$ is the total number of such messages.

Lemma 23. *In any execution of algorithm* AR, $M_d = O(S_a)$.

The remaining messages are categorized into three groups. M_c is the number of messages sent by non designated coordinators during the attended phases plus the number of messages sent in response to such coordinators. M_w is the the number of messages sent by all workers to the expected coordinators during the unattended phases. M_r is the number of messages sent and received by processors that restart during the computation.

Lemma 24. *In any execution of algorithm* AR, $M_c + M_w + M_r = O(f \cdot p)$.

Theorem 25. *In any execution of algorithm* AN, $M = O(t + p \cdot \log p + p \cdot f)$.

5 Discussion

We have considered the Do-All problem of performing t tasks on a distributed system of p fault-prone synchronous processors. We presented the first algorithm for the model with processor failures and restarts. Previous algorithms accommodated only stop-failures. Prior algorithmic approaches relied on the single coordinator paradigm in which the coordinator is elected for the time during which the progress of the computation depends on it. However this approach is not effective in the general model with processor restarts: an omniscient adversary can always stop the single coordinator while keeping alive all other processors thus preventing any global progress. In this paper we have used a novel multi-coordinator paradigm in which the number of simultaneous coordinators increases exponentially in response to coordinator failures. This approach enables effective Do-All solutions that accommodate processor restarts. Moreover, when there are no restarts, the performance of the algorithm is comparable to that of any known algorithm.

The fault-prone processors in our algorithms use reliable communication. It can be shown, for example, that with minor modifications, our algorithms remain correct and efficient even if worker-to-coordinator multicasts are not reliable. However coordinators still need to use reliable broadcast. A worthwhile research direction is to design algorithms which use our aggressive coordinator paradigm and unreliable communication.

Acknowledgments: We thank Moti Yung for several discussions of processor restart issues and for encouraging this direction of research.

References

1. R. De Prisco, A. Mayer, and M. Yung, "Time-Optimal Message-Efficient Work Performance in the Presence of Faults," in *Proc. 13th ACM Symposium on Principles of Distributed Computing*, 1994, pp. 161–172.
2. C. Dwork, J. Halpern, O. Waarts, "Performing Work Efficiently in the Presence of Faults", to appear in *SIAM J. on Computing*, prelim. vers. appeared as Accomplishing Work in the Presence of Failures in *Proc. 11th ACM Symposium on Principles of Distributed Computing*, pp. 91-102, 1992.
3. Z. Galil, A. Mayer, and M. Yung, "Resolving Message Complexity of Byzantine Agreement and Beyond," in *Proc. 36th IEEE Symposium on Foundations of Computer Science*, 1995, pp. 724–733.
4. V. Hadzilacos and S. Toueg, "Fault-Tolerant Broadcasts and Related Problems," in *Distributed Systems*, 2nd Ed., S. Mullender, Ed., Addison-Wesley and ACM Press, 1993.
5. P.C. Kanellakis, D. Michailidis, A.A. Shvartsman, "Controlling Memory Access Concurrency in Efficient Fault-Tolerant Parallel Algorithms", *Nordic J. of Computing*, vol. 2, pp. 146-180, 1995 (prel. vers. in *WDAG-7*, pp. 99-114, 1993).
6. P.C. Kanellakis and A.A. Shvartsman, "Efficient Parallel Algorithms Can Be Made Robust," *Distributed Computing*, vol. 5, pp. 201–217, 1992; prel. version in *Proc. of the 8th ACM Symp. on Principles of Distributed Computing*, 1989, pp. 211–222.
7. P.C. Kanellakis and A.A. Shvartsman, *Fault-Tolerant Parallel Computation*, ISBN 0-7923-9922-6, Kluwer Academic Publishers, 1997.
8. Z.M. Kedem, K.V. Palem, and P. Spirakis, "Efficient Robust Parallel Computations," *Proc. 22nd ACM Symp. on Theory of Computing*, pp. 138-148, 1990.
9. Z.M. Kedem, K.V. Palem, M.O. Rabin, A. Raghunathan, "Efficient Program Transformations for Resilient Parallel Computation via Randomization," in *Proc. 24th ACM Symp. on Theory of Comp.*, pp. 306-318, 1992.
10. C. Martel, personal communication, March, 1991.
11. C. Martel, R. Subramonian, and A. Park, "Asynchronous PRAMs are (Almost) as Good as Synchronous PRAMs," in *Proc. 32d IEEE Symposium on Foundations of Computer Science*, pp. 590-599, 1990.