Distrib. Comput. (2001) 14: 49–64

Performing tasks on synchronous restartable message-passing processors

Bogdan S. Chlebus[1], **Roberto De Prisco**[2,3], **Alex A. Shvartsman**[2,4,*]

[1] Instytut Informatyki, Uniwersytet Warszawski, ul. Banacha 2, 02-097 Warszawa, Poland (e-mail: chlebus@mimuw.edu.pl)
[2] Laboratory for Computer Science, Massachusetts Institute of Technology, 545 Technology Square, NE43-316 Cambridge, MA 02139, USA (e-mail: robdep@theory.lcs.mit.edu, alex@theory.lcs.mit.edu)
[3] Dipartimento di Informatica ed Applicazioni, University of Salerno, 84081 Baronissi (SA), Italy
[4] Department of Computer Science and Engineering, University of Connecticut, 191 Auditorium Road, Unit 155, Storrs, CT 06269, USA

Received: October 1998 / Accepted: September 2000

Abstract. This work considers the problem of performing t tasks in a distributed system of p fault-prone processors. This problem, called DO-ALL herein, was introduced by Dwork, Halpern and Waarts. The solutions presented here are for the model of computation that abstracts a synchronous message-passing distributed system with processor stop-failures and restarts. We present two new algorithms based on a new aggressive coordination paradigm by which multiple coordinators may be active as the result of failures. The first algorithm is tolerant of $f < p$ stop-failures and does not allow restarts. Its available processor steps (work) complexity is $S = \mathcal{O}((t + p \log p / \log\log p) \cdot \log f)$ and its message complexity is $M = \mathcal{O}(t + p \log p / \log\log p + fp)$. Unlike prior solutions, our algorithm uses redundant broadcasts when encountering failures and, for $p = t$ and large f, it achieves better work complexity. This algorithm is used as the basis for another algorithm that tolerates 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 is $S = \mathcal{O}((t + p \log p + f) \cdot \min\{\log p, \log f\})$, and its message complexity is $M = \mathcal{O}(t + p \log p + fp)$, where f is the total number of failures.

Key words: Fault-tolerance – Distributed systems – Load balancing – Processor restarts – Work

A preliminary version of this work appeared as [2]. 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, KBN 8 T11C 036 14, and DFG-Graduiertenkolleg "Parallele Rechnernetzwerke in der Produktionstechnik" ME 872/4-1, DFG-SFB 376 "Massive Parallelität: Algorithmen, Entwurfsmethoden, Anwendungen". The research of the first and the third authors was partly done while visiting Heinz Nixdorf Institut, Universität-GH Paderborn.

* The research of the third author was supported in part by the NSF CAREER Award CCR 9984778 and by the NSF Grant CCR 9988304.

1 Introduction

Achieving efficient distributed solutions for specific problems depends on our ability to effectively exploit parallelism in a system consisting of multiple processors. This is often challenging because the set of processors available to a computation may dynamically change. Such changes may occur due to processor failures or processors becoming unavailable during periods when they are required to perform other unrelated tasks, or due to repaired or idle processors joining the computation already in progress. A basic problem that can readily benefit from adaptively parallel solutions is the problem of performing a number of similar, independent and idempotent tasks. By the similarity of 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 without negatively impacting the final result. Examples of such problems are checking all the points in a large solution space, trying to generate a witness or refute its existence, or simply performing a number of similar independent calculations.

Here we consider the abstract problem of performing t tasks in a synchronous message passing distributed environment consisting of p processors, which are subject to failures and restarts. Failures are crash failures, i.e., a faulty processor stops and does not perform any further actions. Restarted processors resume computation in a predefined initial state, i.e., no stable storage is assumed. We refer to such a problem as the DO-ALL problem.

Algorithmic solutions for the DO-ALL problem in the message-passing models of computation can be evaluated according to their computational effectiveness that measures the number of computation steps taken in performing the tasks, and according to their communication efficiency that measures the amount of communication needed to perform the tasks. Dwork, Halpern and Waarts [6], the first to consider the DO-ALL problem, use a *work* measure defined as the number of tasks executed, counting multiplicities, to assess the computational efficiency. This work measure accounts only for steps taken by processors while executing the tasks of the DO-ALL problem; processor steps taken for coordination or waiting

		S: available processor steps	M: message complexity
No	[5]	$\mathcal{O}(t+(f+1)p)$	$\mathcal{O}((f+1)p)$
restarts	[7]	$\mathcal{O}(t+(f+1)p)$	$\mathcal{O}(fp^{\varepsilon}+\min\{f+1,\log p\}p)$
$(f<p)$	AN	$\mathcal{O}((t+p\log p/\log\log p)\log f)$	$\mathcal{O}(t+p\log p/\log\log p+fp)$
Restarts $(f<p+r)$	AR	$\mathcal{O}((t+p\log p+f)\cdot\min\{\log p,\log f\})$	$\mathcal{O}(t+p\log p+fp)$

Fig. 1. Efficiency of the solutions in [5, 7] and algorithms AN and AR (the solutions in [6] consider a different notion of work complexity and focus on evaluation of effort)

for messages are not counted. Another measure of work, the *available processor steps*, defined by Kanellakis and Shvartsman [10], takes into account all steps taken by the processors, that is, both steps taken in executing the t tasks and any other steps, including idling, taken by the available processors. Thus the available processor steps measure [10] is more conservative than the work measure of [6]. Let $W(t,p)$ be the work complexity and $S(t,p)$ be the available processor steps complexity of some DO-ALL algorithm in some failure model. It is always the case that $W(t,p)=\mathcal{O}(S(t,p))$, since $S(t,p)$ counts the idle/wait steps, which are not included in $W(t,p)$. The equality $W(t,p)=S(t,p)$ can be achieved, for example, by algorithms that perform at least one task during any fixed time period. In our work we use the available processor steps measure.

Communication efficiency is gauged using the message complexity that accounts for the number of messages sent during the computation, or, when the messages substantially vary in size, using the bit complexity that accounts for the number of bits sent. When processors communicate using broadcasts (multicasts), it is possible to measure the communication complexity either in terms of the total number of broadcast messages, or in terms of the number of messages destined to all recipients targetted by the broadcasts. In this work we use the more conservative communication complexity measure by taking into account all messages created as the result of a broadcast. For example, we count a single broadcast to p processors as p messages.

Dwork *et al.* also use the *effort* complexity, defined as the sum of the work and message complexities. This approach makes sense for algorithms for which the work and the message complexities are similar. However, this makes it difficult to compare relative efficiency of algorithms that exhibit varying trade-offs between the work and the communication efficiencies. De Prisco, Mayer and Yung [5] evaluate DO-ALL algorithms using a "lexicographic" criterion: first evaluate an algorithm according to its available processor steps and then according to its message complexity. This approach assumes that optimization of the computational steps is more important than that of the message complexity. In this paper we consider the available processor steps, denoted by S, and the message complexity, denoted by M, as two *independent* measures of efficiency of algorithms.

It is not difficult to formulate trivial solutions to DO-ALL in which each processor performs each of the t tasks. Such solutions have $S=\Omega(t(p+r))$, where r is the number of restarts, and they do not require any communication. Solutions that achieve better efficiency in S trade messages for computation steps.

Review of prior work

Algorithms solving the DO-ALL problem have been provided by Dwork, Halpern and Waarts [6], by De Prisco, Mayer and Yung [5], and by Galil, Mayer and Yung [7]. These deterministic algorithms are formulated for failure models that allow processor failures but disallow processor restarts. The point-to-point messaging between non-faulty processors is assumed to be reliable. In a synchronous system with these assumptions processor failures are detectable, for example using a timeout, and such processors are modeled using the *fail-stop processor* abstraction of Schlichting and Schneider [15].

Dwork, Halpern and Waarts [6] developed the first algorithms for the DO-ALL problem. One algorithm presented in [6] (protocol $\mathcal{B}$) has effort $\mathcal{O}(t+p\sqrt{p})$, with work contributing the cost $\mathcal{O}(t+p)$ towards the effort, and message complexity contributing the cost $\mathcal{O}(p\sqrt{p})$. The running time of the algorithm is $\mathcal{O}(t+p)$. The algorithm uses the synchrony of the system to detect failures by means of time-outs. In this algorithm the t tasks are divided into chunks and each of these 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 in [6] (protocol $\mathcal{C}$) has effort $\mathcal{O}(t+p\log p)$. It has optimal work of $\mathcal{O}(t+p)$, message complexity of $\mathcal{O}(p\log p)$, and time $\mathcal{O}(p^2(t+p)2^{t+p})$. Thus the reduction in message complexity is traded-off for a significant increase in time. Yet another algorithm of [6] (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 $\Theta(f\,p^2)$.

De Prisco, Mayer and Yung [5] present an algorithm which has the available processor steps $\mathcal{O}(t+(f+1)p)$ and message complexity $\mathcal{O}((f+1)p)$. The available processor steps and communication efficiency approach requires keeping all the processors busy doing tasks, simultaneously controlling the amount of communication. Their algorithm operates as follows. At each step all the processors have an overestimate of the set of all the available processors. One processor is designed to be the coordinator and is responsible for the progress of the computation. It allocates the outstanding tasks according to some allocation 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 ($p\ll t$) is assumed.

Another efficient algorithm was developed by Galil, Mayer and Yung [7]. Working in the context of Byzantine agreement with stop-failures (for which they establish a message-optimal solution), they improved the message complexity of [5] to $\mathcal{O}(fp^{\varepsilon}+\min\{f+1,\log p\}p)$, for any positive ε, while achieving the available processor steps complexity of $\mathcal{O}(t+(f+1)p)$.

In [5] a lower bound of $\Omega(t+(f+1)p)$ for algorithms that use the stage-checkpointing strategy is proved, this bound being quadratic in p for f comparable with p. However there are algorithmic strategies that have the potential of circumventing the quadratic bound. Consider the following scenarios. In the first one we have $t = o(p)$, $f > p/2$, and the algorithm assigns all tasks to every processor. Then $S = \mathcal{O}(pt) = o(t + (f + 1)p)$, because $fp = \Theta(p^2)$. This naïve algorithm has a quadratic S for $p = \mathcal{O}(t)$. 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, execution of tasks is interleaved with communication, and the outstanding tasks are evenly allocated among the live processors based on their identifiers. The tasks 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 $S = \mathcal{O}(p \log p/ \log\log p)$. This bound is $o(t + (f + 1)p)$ for $f > p/2$ and $t = p$. Unfortunately the number of messages exchanged is more than quadratic, and can be $\Omega(p^2 \log p/ \log\log p)$. These examples suggest a possibility of performance better than $S = \mathcal{O}(t + (f + 1)p)$, 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.

Previous deterministic algorithms are 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 $S = \Omega(t + (f + 1)p)$. Namely, consider the following behavior of the adversary: while there is more than one operational processor, the adversary stops each coordinator immediately after it becomes one and before it sends any messages. This creates pauses of $\Omega(1)$ steps, giving the $\Omega((f + 1)p)$ part, where f is the number of stop-failures ($f < p$). 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 lower-bound argument for stage-checkpointing strategies is formally presented in [5]. Moreover, when processor restarts are allowed, any algorithm that relies on a single coordinator for information gathering might not terminate, because the adversary can always kill the current coordinator, keeping alive all the other processors so that no progress is made.

Summary of contributions

All previous algorithms do not consider the possibility that a faulty processor is repaired and reintegrated into the system. In this paper we present the first algorithm that solves the DO-ALL problem allowing processor restarts. We introduce a new algorithmic technique based on an aggressive coordination paradigm that permits multiple concurrent coordinators. This approach is suggested by the earlier observation that algorithms with only one coordinator cannot deal efficiently with restarts. The number of coordinators is managed adaptively. When failures of coordinators disrupt the progress of the computation, the number of coordinators is increased; when the failures subside, a single coordinator is appointed. *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[1] $S = \mathcal{O}((t + p \log p/ \log\log p) \log f)$ and message complexity $M = \mathcal{O}(t + p \log p/ \log\log p + fp)$. Algorithm AN is the basis for our second algorithm, called "algorithm AR" (Algorithm with Restarts), which tolerates stop-failures and restarts. Its available processor steps complexity is $S = \mathcal{O}((t + p \log p + f) \cdot \min\{\log p, \log f\})$, and its message complexity is $M = \mathcal{O}(t + p \log p + fp)$, where f is the number of failures. The results are summarized in Fig. 1.

Our algorithm AN is more efficient in terms of S than the algorithms in [5] and [7] when f, p and t are comparable; the algorithm also has efficient message complexity. Algorithms AN and algorithm AR come within a $\log f$ (and $\log p$) factor of the respective lower bounds [10] proved in the context of the shared-memory model of computation for any algorithms that balance loads of surviving processors in each constant-time step.

Our algorithms assume that the communication is reliable. If a processor sends a message to another operational processor and when the message arrives at the destination the processor is still operational, then the message is received. Moreover, if an operational processor sends a multicast message and then fails, then either the message is sent to all destinations or to none at all. Such multicast is received by all operational processors. Prior solutions do not make this assumption, although they do not solve the problem of processor restarts. The availability of reliable multicast 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 such reliable multicast. 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 broadcast LANs (e.g., Ethernet and bypass rings) have the property that if the sender is transmitting a multicast message, then the message is sent to all destination. Of course this does not guarantee that such multicast will be received, however when a processor is unable to receive or process a message, e.g., due to unavailable buffer space or failure of the network interface hardware at the destination, this can be interpreted as a failure

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

of the receiver. From the standpoint of the sender, the availability of hardware-assisted broadcast makes the communication cost of sending a broadcast message comparable to the communication cost of sending a single point-to-point message. However, since multiple receivers may have to process the broadcast message, we are using a conservative cost measure that assumes that the communication 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. [8]) without altering the overall algorithmic approach. Lastly, our approach presents a new venue for optimizing DO-ALL solutions and for beating the $\Omega(t + (f + 1)p)$ lower bound of stage-checkpointing algorithms [5].

We conjecture 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.

For the fail-stop/restart models we assume that a processor loses its state upon a failure and that its state is reset to some known initial state upon a restart. Our algorithms cannot take direct advantage of such a possibility, and it would be interesting to explore the benefits of having stable storage.

We believe that it is important to consider processor restarts in general-purpose distributed computation. For example, important communication services such as group communication systems [4] are in part motivated by the need to re-integrate processors that have either previously failed or were unable to communicate. In this work we make new contributions to the study of complexity of doing work in the presence of failures and restarts.

Other related work

The DO-ALL problem for the shared-memory model of computation was introduced and studied by Kanellakis and Shvartsman [10, 11] (the problem is called WRITE-ALL in that context). Parallel computation using the iterated DO-ALL paradigm is the subject of several subsequent papers, most notably the work of Kedem, Palem and Spirakis [12], Martel, Park and Subramonian [14] and Kedem, Palem, Rabin and Raghunathan [13]. Kanellakis, Michailidis and Shvartsman [9] 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.

Kanellakis and Shvartsman [11] give matching lower and upper bounds on solving the DO-ALL problem for algorithms that are able to choose the best possible assignment of processors to tasks, for example using an oracle. These lower and upper bounds were developed for the shared-memory model of computation, however the bounds apply, verbatim, to the message-passing model (when the oracle is omniscient). For the model with stop-failures, this bound is $t+p\log p/\log\log p$ and for the model with restarts, this bound is $t+p\log p$. A component of the upper bound on work of our algorithms comes within a small multiplicative factor of these bounds. For the algorithm AN this factor is $\log f$, and for the algortihm AR this factor is $\min\{\log p, \log f\}$.

A randomized solution for the DO-ALL problem is presented by Chlebus and Kowalski [3]. Their work is for the model of faults in which an adversary chooses at most $c \cdot p$ processors prior to the start of the computation, for a fixed constant $0 < c < 1$, and then may fail any of these processors at any time, while the remaining processors will stay operational. The randomized algorithm has both the expected available processor steps and message complexity of $\mathcal{O}(t+p\cdot(1+\log^* p - \log^*(p/t)))$, where $\log^*$ is the number of times the $\log$ function has to be applied to its argument to yield the result that is no larger than 1. This is in contrast with the lower bound $\Omega(t+p\cdot\log t/\log\log t)$ on the available processor steps required in the worst case by any deterministic algorithm in this setting.

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 an analysis. Section 4 gives algorithm AR with a proof of its correctness and an analysis. Section 5 concludes with remarks and future work.

2 Model and algorithmic preliminaries

In Sect. 2.1 we describe the distributed setting considered and in Sect. 2.2 we introduce the main ideas underlying our algorithms.

2.1 Model of computation

Distributed setting. We consider a distributed system consisting of a set $\mathcal{P}$ of p processors. 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. Processors communicate by message passing. The distributed system is synchronous and we assume that the processor clocks are globally synchronized. Processor activities are structured in terms of *steps* that have some fixed known constant duration. In each step a processor can either receive messages or perform some local computation or send messages to other processors.

Messaging assumptions. We assume that the underlying network is fully connected, that is, any processor can send messages to any other processor, and that messages are not lost in transit or corrupted. Messages sent within one step are delivered before the end of the next step. Thus we also assume that there is a known upper bound on message delivery time. We assume that reliable multicast [8] is available. With reliable multicast a processor q can send a message to any set $P \subseteq \mathcal{P}$ of processors and all the processors in P that are alive during the entire following step receive the message sent by q. Note that in any step a processor may receive up to $|\mathcal{P}|$ messages (thus we assume that the time needed to process a received message is small compared to the duration of the step). We are not concerned with the size of messages; however, using bit-string set encoding, each message sent by our algorithms contains $\mathcal{O}(\max\{t, p\})$ bits, where t is the number of tasks.

Tasks. We define a *task* to be a computation that can be performed by any processor in one time step and its execution is independent of the execution of any of the other tasks. The tasks are also *idempotent*, i.e., executing a task many times and/or concurrently has the same effect as executing the task once. Tasks are uniquely identified by their task identifiers (TIDs) and the set of TIDs is totally ordered. We denote by $\mathcal{T}$ the set of t tasks and we assume that $\mathcal{T}$ is known to all the processors.

Models of failure. We are using the *fail-stop* processor model [15]. This means that the processors fail by stopping and that in our synchronous setting processor failures can be detected using a timeout. We consider both the case when no restarts are allowed and the case when processors restart after a failure. A processor may stop at any moment during the computation. A failed processor does not receive any messages and does not perform any computation. Messages delivered to a faulty processor are lost. If restarts are allowed, a processor can restart at any point after a failure. We assume that during a single step a faulty processor can restart at most once (e.g., a processor can restart in response to a clock tick). Upon a restart the state of the restarted processor is reset to its initial state, but the processor is aware of the restart. Since an arbitrary time may elapse between the failure of a processor to its restart, the knowledge of the restarted processor may be arbitrarily out of date. Thus we assume a weak model where the processors do not have stable storage that survives a failure. Stable storage could help, for example, for processors to make individual computational progress when an adversary may completely prevent processors from communicating with each other.

It is obvious that if any pattern of failures is allowed, that is, if no restrictions are imposed on the adversary that causes failures, then computational progress can not be guaranteed. For example, if all the processors fail then no progress is possible. Even if processors restart, progress can be prevented. For example, consider the scenario in which a subset of the processors is alive initially, these processors perform some computation, and then they all crash while the processors in the remaining set restart without any possibility of communication between the two sets. Since there is no stable storage, this can be repeated forever without any progress in computation.

We will consider two families of failure models, one that allows failures but no restarts, and another that allows restarts. The failure models impose some restriction on the failure pattern that the adversary can cause. The following definition is used to qualify certain allowable failure patterns.

Definition 2.1. *Let k be a positive integer. A failure pattern is said to be "k-restricted" if during any consecutive k steps $i, i+1, \ldots, i+k-1$ there is at least one processor that is alive during all steps $i, i+1, \ldots, i+k-1$.*

We now define the failure models. Let $\mathcal{F}_{FS}^{(k)}$ be the failure model defined as the set of all failure patterns that are k-restricted, for $k \geq 0$, and have no processor restarts. The family *FS* of *fail-stop* failure models includes all $\mathcal{F}_{FS}^{(k)}$ for non-negative k. Notice that $\mathcal{F}_{FS}^{(0)}$ imposes no restrictions on the failure patterns, that is, all processors can fail in this model. Similarly we define the failure model $\mathcal{F}_{FSR}^{(k)}$ as the set of all failure patterns that are k-restricted, for $k \geq 0$, and that include processor restarts. The family *FSR* of *fail-stop/restart* failure models includes all $\mathcal{F}_{FSR}^{(k)}$ for non-negative k. Also for the fail-stop/restart failure models, $\mathcal{F}_{FSR}^{(0)}$ imposes no restrictions on the failure patterns. With these definitions, we have that, for each k, $\mathcal{F}_{FS}^{(k)} \subseteq \mathcal{F}_{FSR}^{(k)}$, $\mathcal{F}_{FS}^{(k+1)} \subseteq \mathcal{F}_{FS}^{(k)}$, and $\mathcal{F}_{FSR}^{(k+1)} \subseteq \mathcal{F}_{FSR}^{(k)}$. This is because in each case any failure pattern in the subset model is also a failure pattern for the respective superset model, yet the superset models may allow failure patterns not permitted by the respective subsets.

Given a failure pattern, we denote by f the number of failures and by r the number of restarts. For the family *FS* we have that f is bounded from above by p and $r = 0$, while for the family *FSR* we have that $r \leq f < r+p$. We define the *size* of a failure pattern F to be the number of processor failures f, and we denote it by $|F|$. Our complexity results depend on $|F|$, and since it is always the case that $r \leq f$, the main asymptotic results will not involve r.

The DO-ALL problem and termination conditions. First we define the problem.

Definition 2.2. *Given a failure model, for any set $\mathcal{T}$ of tasks and the set $\mathcal{P}$ of processors, the* DO-ALL *problem is to perform all tasks in $\mathcal{T}$.*

What we mean by performing all tasks is that a terminating algorithm that solves the DO-ALL problem must execute all tasks and at least one processor is aware of this fact. In the context of the model that has k-restricted failure patterns this means that if an algorithm exists for this k, then the algorithm may terminate in step τ when each processor that was active and did not fail in steps $\tau-k, \ldots, \tau-1, \tau$ knows that all tasks have been performed.

As we have noted earlier, the DO-ALL problem is not necessarily solvable in each failure model. Let us first look at the fail-stop models. In $\mathcal{F}_{FS}^{(0)}$ no solution is possible: indeed if all processors fail before executing all the tasks in $\mathcal{T}$, then the tasks can never be completed. Clearly we would like to solve the problem as long as at least one processor is alive, that is, as long as $f < p$. By the definition of $\mathcal{F}_{FS}^{(1)}$ we have that the failure patterns allowed by $\mathcal{F}_{FS}^{(1)}$ are exactly those failure patterns with $f < p$. There is a trivial solution that works for $\mathcal{F}_{FS}^{(1)}$: each processors performs all the task in $\mathcal{T}$. This solution, however is not efficient. We provide an efficient algorithm that solves the DO-ALL problem for $\mathcal{F}_{FS}^{(1)}$. The algorithms in [5–7] also work for $\mathcal{F}_{FS}^{(1)}$. Since $\mathcal{F}_{FS}^{(1)}$ is a superset of $\mathcal{F}_{FS}^{(k)}$, for any $k > 1$, the solution for $\mathcal{F}_{FS}^{(1)}$ is also a solution for $\mathcal{F}_{FS}^{(k)}$. (It can be shown that $\mathcal{F}_{FS}^{(1)} = \mathcal{F}_{FS}^{(k)}$ for any $k > 1$, thus no algorithmic advantage can be achieved by increasing k.)

Next we look at the fail-stop/restart failure models. Since $\mathcal{F}_{FS}^{(0)}$ is a subset of $\mathcal{F}_{FSR}^{(0)}$, no solution is possible for $\mathcal{F}_{FSR}^{(0)}$. It is not hard to see that no solution is possible also for $\mathcal{F}_{FSR}^{(1)}$. Indeed a 1-restricted failure pattern requires that at least one processor be alive during any step. However with a stop-failure/restart model this is not sufficient to guarantee progress. As we have remarked before, even if there is always one processor alive progress can be prevented (the scenario in which half of the processors fail while the other half of the processors restart is an example). Hence the best we can hope for is to find a solution for $\mathcal{F}_{FSR}^{(2)}$. We notice that in a k-restricted execution, for $k \geq 2$, it is guaranteed that processors' lifetimes have some overlap and the bigger is k the bigger is the overlap. For $k = 2$ such overlap can be as small as a single step. Hence in order to not lose information about the ongoing computation (such loss, in the absence of stable storage, prevents progress), it is necessary that processors exchange state information during each step. Thus a solution that works for a small k tends to have large message complexity. We provide an efficient algorithm that solves the DO-ALL problem for $\mathcal{F}_{FSR}^{(26)}$. The constant 26 depends on our implementation of the algorithm. With a modest effort the constant can be reduced to 17, as we explain later. Note also that there is a *qualitative* distinction between $\mathcal{F}_{FSR}^{(1)}$ and $\mathcal{F}_{FSR}^{(2)}$: processors' lifetimes may not overlap in the former while they must overlap in the latter. The difference between $\mathcal{F}_{FSR}^{(k)}$ and $\mathcal{F}_{FSR}^{(k+1)}$ when $k \geq 2$ is *quantitative*: in the latter the overlap of processors' lifetimes is one step longer than in the former.

Performance measures. To evaluate the performance of our algorithms we use *available processor steps* and *communication complexity*. The available processor steps is the number of steps taken by all the processors and the communication complexity is the number of point-to-point messages sent. More formally let $\mathcal{F}$ be the set of allowed failure patterns, that is, the failure model considered. For a computation subject to a failure pattern F, $F \in \mathcal{F}$, denote by $p_i(F)$ the number of live processors executing step i and by $m_i(F)$ the number of point-to-point messages sent during step i. For a given problem, if the computation solves the problem by step τ in the presence of the failure pattern F, then the available processor steps complexity S is:

$$S_{p,f} = \max_{F \in \mathcal{F},\ |F| \leq f} \left\{ \sum_{i \leq \tau} p_i(F) \right\},$$

and the communication complexity M is:

$$M_{p,f} = \max_{F \in \mathcal{F},\ |F| \leq f} \left\{ \sum_{i \leq \tau} m_i(F) \right\}.$$

(Recall that in our definitions: (a) all steps of the operational processors are counted, including any idle/waiting time, and (b) a single multicast counts for as many messages as it has recipients.)

2.2 Overview of algorithmic techniques

Both algorithms proceed in a *loop* which is repeated 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 (thus a phase consists of 9 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. We refer to these three step as the *receive* substage, the *compute* substage and the *send* substage.

Coordinators and workers. A processor can be a *coordinator* of a given phase. All processors (including coordinators) are *workers* in a given phase. Coordinators are responsible for recording progress, while workers perform tasks and report on that to the coordinators. In the first phase one processor acts as the coordinator. There may be multiple coordinators in subsequent phases. 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. Whenever at least one coordinator survives a given phase, the number of coordinators for the next phase is reduced to one.

If at least one processor acts as a coordinator during a phase and it completes the phase without failing, we say that the phase is *attended*, the phase is *unattended* otherwise.

Local views. Processors assume the role of coordinator based on their local knowledge. During the computation each processor w maintains a list $L_w = \langle q_1, q_2, ..., q_k \rangle$ of supposed live processors. We call such list a *local view*. The processors in L_w are partitioned into *layers* consisting of consecutive sublists of L_w: $L_w = \langle \Lambda^0, \Lambda^1, ..., \Lambda^j \rangle^2$. The number of processors in layer Λ^{i+1}, for $i = 0, 1, ..., j-1$, is the double of the number of processors in layer Λ^i. Layer Λ^j may contain less processors. When $\Lambda^0 = \langle q_1 \rangle$ the local view can be visualized as a 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 local view, 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 $q_{2^{i+1}-1}$, with the exception of the very last layer that may contain a smaller number of processors. Processors in a local view do not necessarily appear in the order of processor identifiers (restarted processors are appended at the end of the local view).

Example. Suppose that we have a system of $p = 31$ processors. Assume that for a phase ℓ all processors are in the local view of a worker w. in order of processor identifier, and that the view is a tree-like view (e.g., at the beginning of the computation, for $\ell = 0$). If in phase ℓ processors $1, 5, 7, 18, 20, 21, 22, 23, 24, 31$ fail (hence phase ℓ is unattended) and in phase $\ell+1$, processors $2, 9, 15, 25, 26, 27, 28, 29, 30$ fail (phase $\ell+1$ is attended by processor 3), then the view of processor w for phase $\ell+2$ is the one in Fig. 2. If in phase $\ell+2$ processor 3 fails and processors $5, 22, 29, 31$ restart (phase $\ell+2$ is unattended) and in phase $\ell+3$ processors $4, 6$ fail and processors $1, 2, 9$ restart (phase $\ell+3$ is unattended) then the view of processor w for phase $\ell+4$ is the one in Fig. 3.

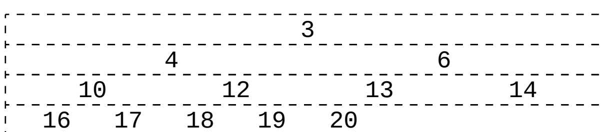

Fig. 2. A local view for phase $\ell+2$.

[2] 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 $\langle e_1, \ldots, e_n, d_1, \ldots, d_m \rangle$.

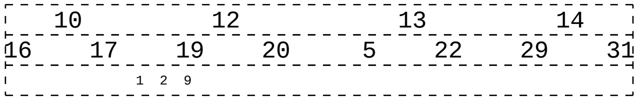

Fig. 3. A local view for phase $\ell + 4$.

The local view is used to implement the martingale principle of appointing coordinators as follows. Let $L_{\ell,w} = \langle \Lambda^0, \Lambda^1, ..., \Lambda^j \rangle$ be the local view of worker w at the beginning of phase ℓ. Processor w expects processors in layer Λ^0 to coordinate phase ℓ; if no processor in layer Λ^0 completes phase ℓ, then processor w expects processors in layer Λ^1 to coordinate phase $\ell + 1$; in general processor w expects processors in layer Λ^i to coordinate phase $\ell + i$ if processors in all previous layers Λ^k, $\ell \le k < \ell + i$, did not complete phase $\ell + k$. The local view is updated at the end of each phase (the update rule depends on the algorithm).

Phase structure and task allocation. The structure of a phase of the algorithms is as follows. Each processor w keeps its local information about the set of tasks already performed, denoted D_w, and the set of live processors, denoted P_w, as known by processor w. The 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" from the start of the phase (here any processors that restarted during the phase are not considered available, since they might not have up to date information about the computation). We denote by U_w the set of *unaccounted* tasks, i.e., the tasks whose done status is unknown to w. The sets U_w and D_w are related by $U_w = \mathcal{T} \setminus D_w$, 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.

Computation starts with phase 0 and any processor q has all processors in $L_{0,q}$ and has $D_{0,q}$ empty. At the beginning of phase ℓ each worker (that is, each processor) w performs one task according to its local view $L_{\ell,w}$ and its knowledge of the set $U_{\ell,w}$ of unaccounted tasks, using the following *load balancing rule*. Worker w executes the task whose rank is $(i \bmod |U_{\ell,w}|)^{th}$ in the set $U_{\ell,w}$ of unaccounted tasks, where i is the rank of processor w in the local view $L_{\ell,w}$. Then the worker reports the execution of the task to all the processors that, according to the worker's local view, are supposed to be coordinators of phase ℓ. For simplicity we assume that a processor sends a message to itself when it is both worker and coordinator. Any processor c that, according to its local view, is supposed to be coordinator, gathers reports from the workers, updates its information about $P_{\ell,c}$ and $U_{\ell,c}$ and broadcasts this new information causing the 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. Restarted processors are reintegrated in the local views and are available for computation in the subsequent phase. A new phase starts if U_ℓ is not empty.

3 Algorithm AN for the fail-stop model

In this section we present, prove correct and analyze algorithm AN which solves the DO-ALL for the failure model $\mathcal{F}^{(1)}_{FS}$.

3.1 *Algorithm* AN

The algorithm follows the algorithm structure described in the previous section. The computation starts with phase number 0 and proceeds in a loop until all tasks are known to have been executed. The detailed description of a phase is given in Fig. 4.

Local view update rule. In phase 0 the local view $L_{0,w}$ of any processor w is a tree-like view containing all the processors in $\mathcal{P}$ ordered by their PIDs. Let $L_{\ell,w} = \langle \Lambda^0, \Lambda^1, ..., \Lambda^j \rangle$ be the local view of processor w for phase ℓ. We distinguish two possible cases.

Case 1. Phase ℓ is unattended. Then the local view of processor w for phase $\ell + 1$ is $L_{\ell+1,w} = \langle \Lambda^1, ..., \Lambda^j \rangle$.

Case 2. Phase ℓ is attended. Then processor w receives `summary` messages from some coordinator in Λ^0. Processor w computes its set P_w as described in stage 3 (we will see that all processors compute the same set P_w). The local view $L_{\ell+1,w}$ of w for phase $\ell+1$ is a tree-like local view containing the processors in P_w ordered by their PIDs.

Figure 6 in Sect. 4 provides a graphical description of a phase of algorithm AN (ignore the messages and steps of restarted processors).

In this section we show that algorithm AN solves the DO-ALL problem for the failure model $\mathcal{F}^{(1)}_{FS}$. Given an execution of the algorithm we say that the execution is *good* if it is an execution allowed by $\mathcal{F}^{(1)}_{FS}$. Hence we have to prove that the algorithm solves the problem for any good execution.

Given an execution of the algorithm, we enumerate the phases. We denote the attended phases of the execution by $\alpha_1, \alpha_2, \ldots$, etc. We denote by π_i the sequence of unattended phases 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. We will show that after a finite number of attended phases the algorithm terminates. If the algorithm correctly solves the problem, it must be the case that there are no tasks left unaccounted after a certain phase α_τ.

Next we show that at the beginning of each phase every live processor has consistent knowledge of the ongoing computation. Then we prove safety (accurate processor and task accounting) and progress (task execution) properties, which imply the correctness of the algorithm.

Lemma 3.1. *In any execution of algorithm* AN, *for any two processors w, v alive at the beginning of phase ℓ, we have that $L_{\ell,w} = L_{\ell,v}$ and that $U_{\ell,w} = U_{\ell,v}$.*

Proof. By induction on the number of phases. For the base case we need to prove that the lemma is true for the first phase. Initially we have that $L_{0,w} = L_{0,v} = \langle \mathcal{P} \rangle$ and $U_w = U_v = \mathcal{T}$. Hence the base case is true.
Assume that the lemma is true for phase ℓ. We need to prove that it is true for phase $\ell + 1$. Let w and v be two processors alive at the beginning of phase $\ell+1$. Since there are no restarts, processors w and v are alive also at the beginning of phase ℓ. By the inductive hypothesis we have that $L_{\ell,w} = L_{\ell,v}$ and

Phase ℓ of algorithm AN:

STAGE 1. RECEIVE: The receive substage is not used.
COMPUTE: In the compute substage, any processor w performs a specific task z according to the load balancing rule.
SEND: In the send substage processor w sends a `report`(z) to any coordinator, that is, to any processor in the first layer of the local view $L_{\ell,w}$.

STAGE 2. RECEIVE: In the receive substage the coordinators gather `report` messages. For any coordinator c, let $z_c^1, \ldots, z_c^{k_c}$ be the set of TIDs received.
COMPUTE: In the compute substage c sets $D_c \leftarrow D_c \cup \bigcup_{i=1}^{k_c}\{z_c^i\}$, and P_c to the set of processors from which c received `report` messages.
SEND: In the send substage, coordinator c multicasts the message `summary`(D_c, P_c) to processors in P_c.

STAGE 3. RECEIVE: During the receive substage `summary` messages are received by live processors. For any processor w, let $(D_w^1, P_w^1), \ldots, (D_w^{k_w}, P_w^{k_w})$ be the sets received in `summary` messages[3] .
COMPUTE: In the compute substage w sets $D_w \leftarrow D_w^i$ and $P_w \leftarrow P_w^i$ for an arbitrary $i \in \{1, \ldots, k_w\}$ and updates its local view L_w as described below.
SEND: The send substage is not used.

Fig. 4. Detailed descrpition of a phase of Algorithm AN.

$U_{\ell,w} = U_{\ell,v}$. We now distinguish two possible cases: phase ℓ is unattended and phase ℓ is attended.

3.2 Correctness of algorithm AN

Case 1. Phase ℓ is unattended. Then there are no coordinators and no `summary` messages are received by w and v during phase ℓ. Thus the sets U_w and U_v are not modified during phase ℓ. Moreover processors w and v use the same rule to update the local view (case 1 of the local view update rule). Hence $L_{\ell+1,w} = L_{\ell+1,v}$ and $U_{\ell+1,w} = U_{\ell+1,v}$.

Case 2. Phase ℓ is attended. Since $L_{\ell,w} = L_{\ell,v}$ all the workers send `report` messages to some coordinators $c_1, ..., c_k$. Since we have reliable multicast, the `report` message of each worker reaches all the coordinators if the worker is alive, or no one if it failed. Thus `summary` messages sent by the coordinators are all the same. Let `summary`(D, P) be one such a message. Since the phase is attended and broadcast is reliable both processors w and v receive the `summary`(D, P) message from at least one coordinator. Hence in stage 3 of phase ℓ, workers w and v set $D_{\ell+1,w} = D_{\ell+1,v} = D$ and consequently we have $U_{\ell+1,w} = U_{\ell+1,v}$. They also set $P_{\ell+1,w} = P_{\ell+1,v} = P$ and use the same rule (case 2 of the local view update rule) to update the local view. Hence $L_{\ell+1,w} = L_{\ell+1,v}$. □

Because of Lemma 3.1, we can define $L_\ell = L_{\ell,w}$ for any live processor w as the view at the beginning of phase ℓ, $P_\ell = P_{\ell,w}$ as the set of live 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.

We denote by p_ℓ the cardinality of the set of live processors computed for phase ℓ, i.e., $p_\ell = |P_\ell|$, and by u_ℓ the cardinality of the set of unaccounted tasks for phase ℓ, i.e., $u_\ell = |U_\ell|$. We have $p_1 = p$ and $u_0 = t$.

[3] As we will see in Sect. 3.2, these messages are in fact identical.

Lemma 3.2. *In any execution of algorithm* AN*, if a processor w is alive during the first two stages of phase ℓ then processor w belongs to P_ℓ.*

Proof. Let w be a processor alive at the beginning of phase ℓ. Processor w (whether it is a coordinator or not) is taken out of the set P_ℓ only if a coordinator does not receive a `report` message from w in phase $\ell - 1$. If w is a coordinator and all coordinators are dead, then w would be removed by the local view update rule. This is possible only if w fails during phase $\ell - 1$. Since w is alive at the beginning of phase ℓ, processor w does not fail in phase $\ell - 1$. □

Lemma 3.3. *In any good execution of algorithm* AN*, if a task z does not belong to U_ℓ then it has been executed in one of the phases* $1, 2, ..., \ell - 1$.

Proof. Task z is taken out of the set U_ℓ by a coordinator c when c receives a `report`(z) message in a phase prior to ℓ. However a worker sends such a message only after executing task z. Task z is taken out of the set U_ℓ by a worker w when w receives a `summary`(D_c, P_c) message from some coordinator c in phase prior to ℓ, and $z \in D_c$. Again this means that z must have been reported as done to c. □

Lemma 3.4. *In any good execution of algorithm* AN*, for any phase ℓ we have that $u_{\ell+1} \leq u_\ell$.*

Proof. By the code of the algorithm, no task is added to U_ℓ. □

Lemma 3.5. *In any good execution of algorithm* AN*, for any attended phase ℓ we have that $u_{\ell+1} < u_\ell$.*

Proof. Since phase ℓ is attended, there is at least one coordinator c alive in phase ℓ. By Lemma 3.2 processor c belongs to P_ℓ and thus it executes one task. Hence at least one task is executed and consequently at least one task is taken out of U_ℓ. By Lemma 3.4, no task is added to U_ℓ during phase ℓ. □

Lemma 3.6. *In a good execution of algorithm* AN*, any unattended period consists of at most* $\log f$ *phases.*

Proof. Consider the unattended period π_i and let ℓ be its first phase. First we claim that the first layer of view L_ℓ consists of a single processor. This is so because (a) either $i = 0$ and $\ell = 0$, in which case L_0 is the initial local view, or (b) $i > 0$ and π_i is preceded by attended phase α_i, in which case L_ℓ is constructed by the local update rule to have a single processor in its first layer. By Lemma 3.2 any processor alive at the beginning of phase ℓ belongs to P_ℓ and thus to L_ℓ. By the local view update rule for unattended phases, we have that eventually all processors in L_ℓ are supposed to be coordinators. Since $f < p$, at least one processor is alive and thus eventually there is an attended phase. The $\log f$ upper bound follows from the the martingale principle governing the sizes of consecutive layers of view. The number of processors accommodated in the layers of the view doubles for each successive layer. Hence, denoting by f_i the number of failures in π_i, we have that the number of phases in π_i is at most $\log f_i$. Obviously $f_i < f$. □

Finally we show the correctness of algorithm AN.

Theorem 3.7. *In a good execution of algorithm* AN*, the algorithm terminates with all tasks performed.*

Proof. By Lemma 3.2 no live processor leaves the computation and since $f < p$ the computation ends only when U_ℓ is empty. By Lemma 3.3, when the computation ends, all tasks are performed. It remains to prove that the algorithm actually terminates. By Lemma 3.6 for every $1 + \log f$ phases there is at least one attended phase. Hence, by Lemmata 3.4 and 3.5, the number of unaccounted tasks decreases by at least one in every $1 + \log f$ phases. Thus, the algorithm terminates after at most $O(t \log f)$ phases. □

Since the algorithm terminates after a finite number of attended phases with all tasks performed, we let τ be such that $U_{\alpha_{\tau+1}} = \emptyset$, and consequently $u_{\alpha_{\tau+1}} = 0$.

3.3 Analysis of algorithm AN

We now analyze the performance of algorithm AN in terms of the available processor steps S and the number of messages M.

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 we have $S = S_a + S_u$.
The following lemma uses the construction by Martel, as it is presented in Lemma 3.3.4 in [10].

Lemma 3.8. *In any good execution of algorithm* AN*we have* $S_a = \mathcal{O}(t + p \log p / \log \log p)$.

Proof. We consider all the attended phases $\alpha_1, \alpha_2, ..., \alpha_\tau$ by subdividing them into two cases.

Case 1. All attended phases α_i such that $p_{\alpha_i} \le u_{\alpha_i}$. The load balancing rule assures that at most one processor is assigned to a task. Hence the available processor steps used in this case can be charged to the number of tasks executed which is at most $t + f \le t + p$. Hence $S_1 = O(t + p)$.

Case 2. All attended phases in which $p_{\alpha_i} > u_{\alpha_i}$. We let $d(p)$ stand for $\log p / \log \log p$. We consider the following two subcases.

Subcase 2.1. All attended phases α_i after which $u_{\alpha_{i+1}} < u_{\alpha_i}/d(p)$. Since $u_{\alpha_{i+1}} < u_{\alpha_i} < p_{\alpha_i} < p$ and phase α_τ is the last phase for which $u_\tau > 0$, it follows that subcase 2.1 occurs $\mathcal{O}(\log_{d(p)} p)$ times. The quantity $\mathcal{O}(\log_{d(p)} p)$ is $\mathcal{O}(d(p))$ because $d(p)^{d(p)} = \Theta(p)$. No more than p processors complete such phases, therefore the part $S_{2.1}$ of S_a spent in this case is

$$S_{2.1} = \mathcal{O}\left(p \frac{\log p}{\log \log p}\right).$$

Subcase 2.2. All attended phases α_i after which $u_{\alpha_{i+1}} \ge u_{\alpha_i}/d(p)$. Consider a particular phase α_i. Since in this case $p_{\alpha_i} > u_{\alpha_i}$, by the load balancing rule at least $\lfloor \frac{p_{\alpha_i}}{u_{\alpha_i}} \rfloor$ but no more than $\lceil \frac{p_{\alpha_i}}{u_{\alpha_i}} \rceil$ processors are assigned to each of the u_{α_i} unaccounted tasks. Since $u_{\alpha_{i+1}}$ tasks remain unaccounted after phase α_i, the number of processors that failed during this phase is at least

$$\begin{aligned} u_{\alpha_{i+1}} \left\lfloor \frac{p_{\alpha_i}}{u_{\alpha_i}} \right\rfloor &\ge \frac{u_{\alpha_i}}{d(p)} \cdot \frac{p_{\alpha_i}}{2u_{\alpha_i}} \\ &= \frac{p_{\alpha_i}}{2d(p)} \; . \end{aligned}$$

Hence, the number of processors that proceed to phase α_{i+1} is no more than

$$p_{\alpha_i} - \frac{p_{\alpha_i}}{2d(p)} = p_{\alpha_i}(1 - \frac{1}{2d(p)}) \; .$$

Let $\alpha_{i_0}, \alpha_{i_1}, ..., \alpha_{i_k}$ be the attended phases in this subcase. Since the number of processor in phase α_{i_0} is at most p, the number of processors alive in phase α_{i_j} for $j > 0$ is at most $p(1 - \frac{1}{2d(p)})^j$. Therefore the part $S_{2.2}$ of S_a spent in this case is bounded as follows:

$$\begin{aligned} S_{2.2} &\le \sum_{j=0}^{k} p \left(1 - \frac{1}{2d(p)}\right)^j \\ &\le \frac{p}{1 - (1 - \frac{1}{2d(p)})} \\ &= p \cdot 2d(p) \\ &= \mathcal{O}(p \cdot d(p)) \; . \end{aligned}$$

Summing up the contributions of all the cases considered we get S_a:

$$S_a = S_1 + S_{2.1} + S_{2.2} = \mathcal{O}\left(t + p \frac{\log p}{\log \log p}\right).$$

□

Lemma 3.9. *In any good execution of algorithm* AN *we have* $S_u = \mathcal{O}(S_a \log f)$.

Proof. The number of processors alive in a phase of the unattended period π_i is at most p_{α_i}, that is the number of processors alive in the attended phase immediately preceding π_i. To cover the case when π_0 is not empty, we let $\alpha_0 = 0$ and $p_{\alpha_0} = |\mathcal{P}| = p$. By Lemma 3.6 the number of phases in period

π_i is at most $\log f$. Hence the part of S_u spent in period π_i is at most $p_{\alpha_i} \log f$. We have

$$\begin{aligned} S_u &\le \sum_{i=0}^{\tau}(p_{\alpha_i} \log f) \\ &= \log f \cdot \sum_{i=1}^{\tau} p_{\alpha_i} \\ &\le (p + S_a) \log f = \mathcal{O}(S_a \log f) . \end{aligned}$$

□

Theorem 3.10. *In any good execution of algorithm* AN *we have* $S = \mathcal{O}(\log f (t + p \log p/ \log \log p))$.

Proof. The total available processor steps S is given by $S = S_a + S_u$. The theorem follows from Lemmata 3.8 and 3.9. □

Remark. A lower bound of $\Omega(t+p \log p/ \log \log p)$ [10] (Theorem 4.2.4) is known for any algorithm that performs tasks by balancing loads of surviving processors in each time step. Although that lower bound was derived for the shared-memory model of computation, the result does not use any arguments involving shared-memory. The work of algorithm AN comes within a factor of $\log f$ (and thus also $\log p$) relative to that lower bound. This suggests that improving the work result is difficult and that better solutions may have to involve a trade-off between the work and message complexities. *Kramer.*

We now assess the message complexity. First remember that the computation proceeds as follows: $\pi_0, \alpha_1, \pi_1, \alpha_2, ..., \pi_{\tau-1}, \alpha_\tau$. In order to count the total number of messages we distinguish between the attended phases preceded by a nonempty unattended period and the attended phases which are not preceded by unattended periods. Formally, we let M_u be the number of messages sent in $\pi_{i-1}\alpha_i$, for all those i's such that π_{i-1} is nonempty and we let M_a be the number of messages sent in $\pi_{i-1}\alpha_i$, for all those i's such that π_{i-1} is empty (clearly in these cases we have $\pi_{i-1}\alpha_i = \alpha_i$). Next we estimate M_a and M_u and thus the message complexity M of algorithm AN.

Lemma 3.11. *In any execution of algorithm* AN *we have* $M_a = O(t + p \log p/ \log \log p)$.

Proof. First notice that in a phase ℓ where there is a unique coordinator the number of messages sent is $2p_\ell$. By the definition of M_a, messages counted in M_a are messages sent in a phase α_i such that π_{i-1} is empty. This means that the phase previous to α_i is α_{i-1} which, by definition, is attended. Hence by the local view update rule of attended phases we have that α_i has a unique coordinator. Thus phase α_i gives a contribution of at most $2p_{\alpha_i}$ messages to M_a. It is possible that some of the attended phases do not contribute to M_a, however counting all the attended phases as contributing to M_a we have that $M_a \le \sum_{i=1}^{\tau} 2p_{\alpha_i} = 2S_a$. The lemma follows from Lemma 3.8. □

Lemma 3.12. *In any good execution of algorithm* AN *we have* $M_u = O(fp)$.

Proof. First we notice that in any phase the number of messages sent is $O(cp)$ where c is the number of coordinators for that phase. Hence to estimate M_u we simple count all the supposed coordinators in the phases included in $\pi_{i-1}\alpha_i$, where π_{i-1} is nonempty.

Let i be such that π_{i-1} is not empty. Since the number of processors doubles in each consecutive layer of the local view according to the martingale principle, we have that the total number of supposed coordinators in all the phases of $\pi_{i-1}\alpha_i$ is $2f_{i-1} + 1 = O(f_{i-1})$, where f_{i-1} is the number of failures during π_{i-1}. Hence the total number of supposed coordinators, in all of the phases contributing to M_u, is $\sum_{i=1}^{\tau} O(f_{i-1}) = O(f)$.
Hence the total number of messages counted in M_u is $O(fp)$.
□

Theorem 3.13. *In any good execution of algorithm* AN *the number of messages sent is* $M = \mathcal{O}(t + p \log p/ \log \log p + fp)$.

Proof. The total number of messages sent is $M = M_a + M_u$. The theorem follows from Lemmata 3.11 and 3.12. □

4 Algorithm AR for the fail-stop/restart model

In this section we present, prove correct and analyze algorithm AR which solves the DO-ALL for the failure model $\mathcal{F}_{FSR}^{(26)}$.

4.1 Algorithm AR

Algorithm AR is similar to algorithm AN; the difference is that there are added messages to handle the restarts of processors. In Fig. 5 we provide the detailed description for each stage of a phase. The parts that are new or that are different in algorithm AR as compared to algorithm AN are *italicized*.

After the restart, processor q broadcasts `restart`(q) messages in each step until it receives a response. Processors receiving such messages, ignore them if these messages are not received in the receive substage of stage 2 of a phase. Thus we can imagine that a restarted processor q broadcasts a `restart`(q) in the send substage of stage 1 of a phase ℓ (however we will count all the `restart` messages in the message complexity). 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 phase $\ell+1$. Processor q needs to be informed about the status of the ongoing computation. Hence processors that have this information send the `info`(U_ℓ, L_ℓ) messages to processor q with the set U_ℓ of unaccounted tasks and the local view L_ℓ.

Loal view update rule. In phase 0 the local view $L_{0,w}$ of any processor w contains all the processors in $\mathcal{P}$ ordered by their PIDs, and the first layer is a singleton set. Let $L_{\ell,w} = \langle \Lambda^0, \Lambda^1, ..., \Lambda^j \rangle$ be the local view of processor w for phase ℓ. We distinguish two possible cases.

Case 1. Phase ℓ is unattended. Let R^ℓ be the set of restarted processors which send `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

Phase ℓ of algorithm AR**:**

STAGE 1. RECEIVE: The receive substage is not used.
COMPUTE: In the compute substage any processor w performs a specific task z according to the load balancing rule.
SEND: In the send substage w sends a `report`(z) to any coordinator, that is, to any processor in the first layer of $L_{\ell,w}$. *Any restarted processor q broadcasts the* `restart`*(q) message informing all live processors of its restart.*

STAGE 2. RECEIVE: In the receive substage the coordinators gather `report` messages *and all processors gather* `restart` *messages. Let R be the set of processors that sent a* `restart` *message.* For any coordinator c, let $z_c^1, ..., z_c^{k_c}$ be the set of TIDs received in `report` messages.
COMPUTE: In the compute substage c sets $D_c \leftarrow D_c \cup \bigcup_{i=1}^{k_c}\{z_c^i\}$ and P_c to the set of processors from which c received `report` messages.
SEND: In the send substage, coordinator c multicasts the message `summary`(D_c, P_c) to the processors in P_c *and R. Any processor in P_c sends the message* `info`*(U_ℓ, L_ℓ) to processors in R.*

STAGE 3. RECEIVE: *In the receive substage processors in R receive* `info`(U_ℓ, L_ℓ) *messages and processors in P_c and R receive* `summary`*(D_c, P_c) messages.*
COMPUTE: *In the compute substage, a restarted processor q sets $L_{\ell,q} \leftarrow L_\ell$ and $U_{\ell,q} \leftarrow U_\ell$.* 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. Processor w sets $D_w \leftarrow D_w^i$ and $P_w \leftarrow P_w^i$ for an arbitrary $i \in 1, ..., k_w$ and updates its local view $L_{\ell,w}$ *as described below.*
SEND: The send substage is not used.

Fig. 5. Descrpition of details of one phase of Algorithm AR (the code that differs from Algorithm AN is given in *italics*).

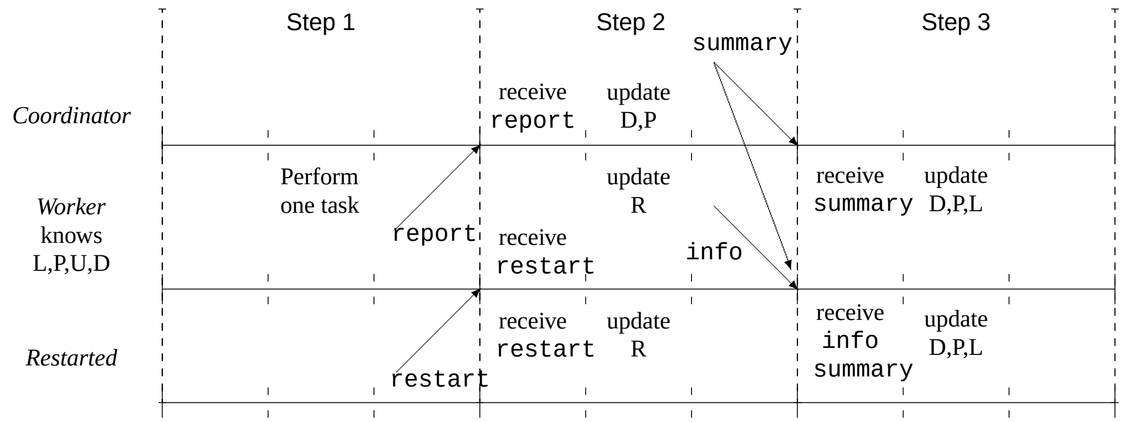

Fig. 6. A phase of algorithm AR (for algorithm AN ignore the bottom line, which represents restarted processors, and all the messages referring to it).

their PIDs. The local view for the next phase is $L_{\ell+1,w} = \langle \Lambda^1, ..., \Lambda^j \rangle \oplus \langle R' \rangle$. The operator $\oplus$ places processors of R', in the order $\langle R' \rangle$, into the last layer Λ^j till this layer contains exactly the double of the processors of layer Λ^{j-1} and possibly adds a new layer Λ^{j+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 view. Notice that restarted processors, which receive `info` messages, know the old view L_ℓ.

Case 2. Phase ℓ is attended. Let R^ℓ be the set of restarted processors. Since the phase is attended `summary` messages are received by all the live processors (including the restarted ones). Any processor w updates P_w as described in stage 3. Processor w knows the set R^ℓ. The local view $L_{\ell+1,w}$ for the next phase is structured according to the martingale principle and contains all the processors in $P_w \cup R^\ell$ ordered according to their PIDs.

If there are no restarts, algorithm AR behaves as algorithm AN. Figure 6 provides a graphical description of both algorithms.

4.2 Correctness of algorithm AR

In this section we show that algorithm AN solves the DO-ALL problem for the failure model $\mathcal{F}_{FSR}^{(26)}$. Given an execution of the algorithm we say that the execution is *good* if it is an execution allowed by $\mathcal{F}_{FSR}^{(26)}$. Hence we have to prove that the algorithm solves the problem for any good execution.

A restarted processor has no information about the ongoing computation, and thus cannot actively participate in the computation, until it gets a chance to communicate with other processors. Moreover, if a processors completes two consecutive phases it is able to acquire information about the computation in the first of the two phases and to transfer it to other processors in the second of the two phases. We will show that having, at any point during any execution, a processor that is operational for 26 consecutive steps is sufficient for our algorithm. This allows for the largest number of steps, 8, that may be "wasted" because this is just short of the 9 steps that constitute a phase, plus two complete phases, i.e., 18 steps, as described above. This intuition is made formal in the proofs in this section.

Formally we use the following definitions.

Definition 4.1. *A live processor is said to be "fully active" at a particular time t during phase ℓ, if it stays alive from the start of phase $\ell - 1$ through time t.*

Definition 4.2. *A live processor is said to be a "witness" for phase ℓ if it stays alive for the duration of phases $\ell - 1$ and ℓ.*

We remark that the difference between a processor fully active in phase ℓ and a witness of phase ℓ is that the witness is guaranteed, by definition, to survive the entire phase ℓ, while the fully active processor may fail before the end of phase ℓ. Hence a fully active processor cannot guarantee transfer of state information while the witness can.

Lemma 4.1. *In a good execution, there is a witness for any phase.*

Proof. A good execution has a 26-restricted failure pattern. Thus for any step i, there is at least one processor that stays alive for the next 26 steps. Notice that 8 of these step may be spent waiting for the beginning of the next phase (if the processor has just restarted in step i). However the remaining 18 steps are enough to guarantee that the processor stays alive for the next two phases, since each phase consists of 9 steps. □

The witness of phase ℓ is always a processor fully active in phase ℓ. Next we show that at the beginning of each phase every fully active processor has consistent knowledge of the ongoing computation.

Lemma 4.2. *In a good execution of algorithm* AR*, for any two processors w, v fully active at the beginning of phase ℓ, we have that $L_{\ell,w} = L_{\ell,v}$ and that $U_{\ell,w} = U_{\ell,v}$.*

Proof. By induction on the number of phases. For the base case we need to prove that the lemma is true for the first phase. Initially we have that $L_{0,w} = L_{0,v} = \langle \mathcal{P} \rangle$ and $U_w = U_v = \mathcal{T}$. Hence the base case is true.
Assume that the lemma is true for phase ℓ. We need to prove that it is true for phase $\ell + 1$. Let w and v be two processors fully active at the beginning of phase $\ell + 1$.
First we claim that at the beginning of stage 3 of phase ℓ, we have $L_{\ell,w} = L_{\ell,v}$ and $U_{\ell,w} = U_{\ell,v}$. Indeed, if w and v are fully active also at the beginning of phase ℓ, then the claim follows by the inductive hypothesis. If processor w (resp. v) has just restarted and is not yet fully active in phase ℓ, then it sends a `restart` message in stage 1 of phase ℓ. By Lemma 4.1, there is a witness for phase ℓ. Hence processor w (resp. v) receives a `info` message from the witness and thus at the beginning of stage 3 of phase ℓ it has $U_{\ell,w} = U_\ell$ (resp. $U_{\ell,v} = U_\ell$) and $L_{\ell,w} = L_\ell$ (resp. $L_{\ell,v} = L_\ell$).
We now distinguish two cases: phase ℓ is attended and phase ℓ is unattended.

Case 1. Phase ℓ is not attended. Then no `summary` messages are received by w and v and in stage 3 of phase ℓ they do not modify their sets $U_{\ell,w}$ and $U_{\ell,v}$. The local view of both processors is modified in the same way (case 1 of the local view update). Hence we have that $U_{\ell+1,w} = U_{\ell+1,v}$ and $L_{\ell+1,w} = L_{\ell+1,v}$.

Case 2. Phase ℓ is attended. Then there is at least one coordinator completing the phase. Let $c_1, ..., c_k$ be the coordinators for phase ℓ. Since we have reliable multicast, the `report` message of each worker reaches all coordinators that are alive. Thus the `summary` messages sent by coordinators are all equal. Let `summary`(D, P) one such a message. Since we have reliable multicast, both processors w and v receive `summary`(D, P) messages from the coordinators. Hence in stage 3 of phase ℓ processors w and v set $D_{\ell+1,w} = D_{\ell+1,v} = D$ and thus we have $U_{\ell+1,w} = U_{\ell+1,v}$. Processors w and v also set $P_{\ell+1,w} = P_{\ell+1,v} = P$ and use the same rule (case 2 of the local view update rule) to update the local view. Hence we have $L_{\ell+1,w} = L_{\ell+1,v}$. □

Because of the previous lemma we can define the view $L_\ell = L_{\ell,w}$, the set of available processors $P_\ell = P_{\ell,w}$, the set of done tasks $D_\ell = D_{\ell,w}$ and the set of unaccounted tasks $U_\ell = U_{\ell,w}$, all of them referred to the beginning of phase ℓ, where w is any fully active processor. Notice that restarted (non-fully-active) processors may have inconsistent knowledge of these quantities.

Remember that we denote by p_ℓ the cardinality of the set of live processors for phase ℓ, i.e., $p_\ell = |P_\ell|$, and by u_ℓ the cardinality of the set of unaccounted tasks for phase ℓ, i.e., $u_\ell = |U_\ell|$.

In the following lemmata we prove safety (no live processor or undone task is forgotten) and progress (tasks execution) properties, which imply the correctness of the algorithm.

Lemma 4.3. *In any execution of algorithm* AR*, a processor fully active at the beginning of phase ℓ belongs to P_ℓ.*

Proof. If processor w is fully active at the beginning of phase $\ell - 1$, then by the inductive hypothesis it belongs to $P_{\ell-1}$. Processor w is taken out of the set P_ℓ only if a coordinator does not receive a `report` message from w in phase $\ell - 1$. Since processor w survives phase $\ell - 1$ then it sends the `report` message in phase $\ell - 1$. Hence it belongs to P_ℓ.
If processor w is not fully active at the beginning of phase $\ell - 1$, then it restarted in phase $\ell - 1$. Thus at the end of phase $\ell - 1$ processor w is re-integrated in the local views of phase ℓ. Hence it belongs to P_ℓ. □

Lemma 4.4. *In any execution of algorithm* AR*, if a task z does not belong to U_ℓ then it has been executed in phases $1, 2, ..., \ell - 1$.*

Proof. The proof is the same as the proof of Lemma 3.3. □

Lemma 4.5. *In a good execution of algorithm* AR*, for any phase ℓ we have that $u_{\ell+1} \leq u_\ell$.*

Proof. Consider phase ℓ. If there are no restarts, then, by the code, no task is added to the set of undone tasks. If there are restarts, a restarted processor w has $U_{\ell,w} = \mathcal{T}$. By Lemma 4.1, there is a processor v which is a witness for phase ℓ. Then processor w receives the `info`(U_ℓ, L_ℓ) message from processor

v and hence sets $U_{\ell,w} = U_\ell$. Hence also when processors restart no task is added to the set of undone tasks. □

Lemma 4.6. *In any good execution of algorithm* AR, *for any attended phase ℓ we have that $u_{\ell+1} < u_\ell$.*

Proof. Since phase ℓ is attended, there is at least one coordinator c alive in phase ℓ. A coordinator must be a fully active processor (a restarted processor needs to complete a phase in order to known the current view and become coordinator). By Lemma 4.3 processor c belongs to P_ℓ and thus it executes one task. Hence at least one task is executed and consequently at least one task is taken out of U_ℓ. By Lemma 4.5, no task is added to U_ℓ during phase ℓ. □

As for algorithm AN, given a particular execution, we denote by $\alpha_1, \alpha_2, ..., \alpha_\tau$ the attended phases and by π_i the unattended period in between phases α_i and α_{i+1}.

Lemma 4.7. *In a good execution of algorithm* AR *any unattended period consists of at most* $\min\{\log p, \log f\}$ *phases.*

Proof. Consider the unattended period π_i. As argued in Lemma 3.6 the views at the beginning of π_i is a tree-like view.
By Lemma 4.3 and by the local view update rule for unattended phases, any processor fully active at the beginning of a phase ℓ of π_i belongs to P_ℓ and thus to L_ℓ. By the local view update rule for unattended phases, we have that eventually there is a phase ℓ' such that all fully active processors are supposed to be coordinators of phase ℓ' (that is, the first layer of $L_{\ell'}$ contains all the processors fully active at the beginning of phase ℓ'). By Lemma 4.1, phase ℓ' has a witness. The witness is a fully active processor and by definition it survives the entire phase. Hence, phase ℓ' is attended.
The upper bounds on the number of phases follow from the tree-like structure of the views. With the same argument used in Lemma 3.6 we have that the number of phases of π_i is at most $\log f$. The $\log p$ bound follows from the fact that by doubling the number of expected coordinators for each unattended phase, after at most $\log p$ phases all processors are expected to be coordinators and thus at least one of them (the witness) survives the phase. □

Theorem 4.8. *In a good execution of algorithm* AR *the algorithm terminates and all the units of work are performed.*

Proof. By Lemma 4.3 fully active processors are always part of the computation, so the computation never ends if there are fully active processors and U_ℓ is not empty. By Lemma 4.1 any phase has a witness which is a fully active processor. The local knowledge about the outstanding tasks is sound, by Lemma 4.4. For every $1+\log p$ phases there is at least one attended phase, by Lemma 4.7. Hence, by Lemmata 4.5 and 4.6, the number of unaccounted tasks decreases by at least one in every $1+\log p$ phases. Thus after at most $\mathcal{O}(t \log p)$ phases all the tasks have been performed. During the next attended phase this information is disseminated and the algorithm terminates. □

4.3 Analysis of algorithm AR

We next analyze the performance of algorithm AR in terms of the available processor steps S used and the number M of messages sent. To assess S we partition it into S_a spent during the attended phases and S_u spent during the unattended phases. So $S = S_a + S_u$. In the following lemmata we assess the available processor steps of algorithm AR.

Recall that good executions are those executions whose failure pattern is allowed by $\mathcal{F}_{FSR}^{(26)}$. We also recall that α_1, $\alpha_2, ..., \alpha_\tau$ denote the attended phases, π_i denote the unattended period in between phases α_i and α_{i+1} and that p_ℓ and u_ℓ denote, respectively, the size of the set P_ℓ of fully active processors for phase ℓ and the size of the set U_ℓ of undone tasks for phase ℓ.

Lemma 4.9. *In a good execution of algorithm* AR *we have* $S_a = \mathcal{O}(t + p\log p + f)$.

Proof. By Theorem 4.8 the algorithm terminates.
We first account for all those steps spent by a processor after a restarts and before the processor either fails again or becomes fully active, that is, it is included in the set P_ℓ for a phase ℓ, and thus is counted for in p_ℓ. The number of such steps spent for each restart is bounded by a constant. Hence the available processor steps spent is $\mathcal{O}(r)$, which is $\mathcal{O}(f)$.
Next we account for all the remaining part of S_a by distinguishing two possible cases:

Case 1. All attended phases α_k such that $p_{\alpha_k} \le u_{\alpha_k}$. The load balancing rule assures that at most one processor is assigned to a task. Hence the available processor steps used in this case can be charged to the number of tasks executed, which is at most $t + f$.

Case 2. All attended phases such that $p_{\alpha_k} > u_{\alpha_k}$. We arrange the tasks that were executed and accounted for during such phases in the order by the phase in which they are performed (for tasks executed in the same phase the order does not matter). Let $\langle b_1, b_2, \ldots, b_m\rangle$ be such a list. Notice that $m \le p$ because $u_{\alpha_k} < p_{\alpha_k} \le p$, and once the inequality $u_{\alpha_k} \le p$ starts to hold, it remains true in phases α_i for $i \ge k$. We then partition these tasks into disjoint adjacent segments Z_i:

$$Z_i = \left\{ b_k : \frac{p}{i+1} \le m-k+1 < \frac{p}{i} \right\}.$$

By the load balancing rule, at most

$$\frac{p}{m-k+1} \le p\frac{i+1}{p} = i+1$$

processors are assigned to each task in Z_i, because when a processor is assigned for the last time to task b_k, there are at least $m-k+1$ unaccounted tasks. The size of Z_i can be estimated as follows:

$$\begin{aligned} |Z_i| &\le \frac{p}{i} - \frac{p}{i+1} \\ &\le p\left(\frac{1}{i} - \frac{1}{i+1}\right) \\ &= \frac{p}{i(i+1)}. \end{aligned}$$

Hence the available processor steps used is less than

$$\sum_{1\le i\le m} \frac{p}{i(i+1)} \cdot (i+1) \le p \sum_{1\le i\le p} \frac{1}{i} = \mathcal{O}(p\log p) \ .$$

Combining all the cases we obtain $S_a = \mathcal{O}(t + p\log p + f)$. □

Lemma 4.10. *In a good execution of algorithm* AR *we have* $S_u = \mathcal{O}(S_a + f) \cdot \min\{\log p, \log f\})$.

Proof. Consider the unattended period π_i. At the beginning of this period there are p_i available processors. By Lemma 4.7, for each of these processors we need to account for $\min\{\log p, \log f\}$ steps spent in period i. Summing up over all attended phases, we have that the part of S_u for these processors is

$$\min\{\log p, \log f\} \cdot \sum_{i=1}^{\tau} p_{\alpha_i} = S_a \cdot \min\{\log p, \log f\}.$$

Each encountered restart can contribute additionally at most $\min\{\log p, \log f\}$ processor steps because if the processor stays alive past phase α_{i+1}, its contribution is already accounted for. Since the number of restarts r is $r \le f$, the bound follows. □

Theorem 4.11. *In a good execution of algorithm* AR *the available processor steps is* $S = \mathcal{O}((t + p\log p + f) \cdot \min\{\log p, \log f\})$.

Proof. The available processor steps S of algorithm AR is given by $S = S_a + S_u$. The theorem follows from Lemmata 4.10 and 4.9. □

Remark. A lower bound of $\Omega(t + p\log p)$ [1] is known for any algorithm that performs tasks by balancing loads of surviving processors in each time step. Although that lower bound was derived for the shared-memory model of computation, the result does not use any arguments involving shared-memory. The work of algorithm AR includes a contribution that comes within a factor of $\min\{\log p, \log f\}$ relative to that lower bound. As we have similarly remarked for algorithm AN, this suggests that improving the work result is difficult and that better solutions may have to involve a trade-off between the work and message complexities. □

We now assess the message complexity. The analysis is similar to the one done for algorithm AN. The difference is that we need to account also for messages sent by restarted processors. However the approach used to analyze the message complexity of algorithm AN works also for algorithm AR.

We distinguish between the attended phases preceded by a nonempty unattended period and the attended phases not preceded by unattended periods. We let M_u be the number of messages sent in $\pi_{i-1}\alpha_i$, for all those i's such that π_{i-1} is nonempty and we let M_a be the number of messages sent in $\pi_{i-1}\alpha_i$, for all those i's such that π_{i-1} is empty (clearly in these cases we have $\pi_{i-1}\alpha_i = \alpha_i$). Next we estimate M_a and M_u and thus the message complexity M of algorithm AR.

Lemma 4.12. *In a good execution of algorithm* AR *we have* $M_a = O(t + p\log p/\log\log p + f)$.

Proof. We first account for messages sent by restarted processors and responses to those messages. For each restart the number of `restart` messages sent is bounded by a constant and one `info` and one `summary` message are sent to a restarted processor before it becomes fully active. Hence the total number of messages sent due to restarts is $\mathcal{O}(r) = \mathcal{O}(f)$.

The remaining messages can be estimated as detailed in Lemma 3.11. In a phase ℓ where there is a unique coordinator the number of messages sent is $2p_\ell$. By the definition of M_a, messages counted in M_a are messages sent in a phase α_i such that π_{i-1} is empty. This means that the phase previous to α_i is α_{i-1} which, by definition, is attended. Hence by the local view update rule of attended phases we have that α_i has a unique coordinator. Thus phase α_i gives a contribution of at most $2p_{\alpha_i}$ messages to M_a. Hence $M_a \le \sum_{i=1}^{\tau} 2p_{\alpha_i} = 2S_a$. The lemma follows from Lemma 4.9. □

Lemma 4.13. *In any good execution of algorithm* AR *we have* $M_u = \mathcal{O}(fp)$.

Proof. We first account for messages sent by restarted processors and responses to those messages. The argument is the same as in Lemma 4.12. The total number of messages sent because of restarts is $O(f)$.

Next we estimate the remaining messages as done in Lemma 3.12. First we notice that in any phase the number of messages sent is $O(cp)$ where c is the number of coordinators for that phase. Hence to estimate M_u we simple count all the supposed coordinators in the phases included in $\pi_{i-1}\alpha_i$, where π_{i-1} is nonempty.
Let i be such that π_{i-1} is not empty. Because of the structure of the local view, we have that the total number of supposed coordinators in all the phases of $\pi_{i-1}\alpha_i$ is $2f_{i-1} + 1 = \mathcal{O}(f_{i-1})$ where f_{i-1} is the number of failures during π_{i-1}. Hence the total number of supposed coordinators, in all of the phases contributing to M_u, is $\sum_{i=1}^{\tau} \mathcal{O}(f_{i-1}) = \mathcal{O}(f)$.
Thus M_u is $\mathcal{O}(fp)$. □

Theorem 4.14. *In a good execution of algorithm* AR *the number of messages sent is* $M = \mathcal{O}(t + p\log p + fp)$.

Proof. The total number of messages sent is $M = M_a + M_u$. The theorem follows from Lemmata 4.12 and 4.13. □

5 Discussion

We have considered the DO-ALL problem which consists 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 do not allow processor restarts. 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 previous algorithms.

There are two areas where improvements can be sought. It appears not difficult to show that in our algorithms worker-to-coordinator multicasts need not be reliable. A worthwhile research direction is to design algorithms which use our aggressive coordinator paradigm and unreliable coordinator-to-worker communication. It is also interesting to consider the models where processors have some stable storage. This may help reduce the reliance on broadcasts as the sole means for information propagation.

For the fail-stop/restart model we developed an algorithm which tolerates failure/restart patterns that are 26-restricted; a 26-restricted failure pattern is one such that for any 26 consecutive steps of the algorithm there is at least one processor alive in all the 26 steps. The constant 26 depends on the algorithm. We conjecture that our algorithm can be easily modified by "squeezing" the phase into two stages, instead of the three used in the presentation for the sake of clarity. With this modification 17-restricted failure patterns can be tolerated. A different approach may solve the problem for k-restricted executions with a smaller k. However the problem is not solvable for 1-restricted executions and, as remarked in Sect. 2, there is a qualitative difference between 1-restricted executions and k-restricted executions, with $k \geq 2$. It is also clear that in order to achieve solutions that work for k-restricted executions for small k it is necessary to use more messages. For example for 2-restricted executions there must be transfer of state information in each step.

Finally, it is also interesting to consider the failure models where k-restriction is imposed not on at least one processor as we have done, but on at least q processors, where q is a failure model parameter. Such definition yields families of failure models $\mathcal{F}_{FS}^{(k,q)}$ and $\mathcal{F}_{FSR}^{(k,q)}$, and more efficient algorithms could be sought for these models. This is because the failure models are more benign, i.e., $\mathcal{F}_{FS}^{(k,1)} \supseteq \mathcal{F}_{FS}^{(k,q)}$ and $\mathcal{F}_{FSR}^{(k,1)} \supseteq \mathcal{F}_{FSR}^{(k,q)}$ for $q > 1$.

Acknowledgements. We thank Moti Yung for several discussions of processor restart issues and for encouraging this direction of research. We thank Greg Malewicz and Chryssis Georgiou for several helpful observations. Finally, we thank the anonymous referees for many comments that had enabled us to improve the quality of the presentation.

References

1. J. Buss, P.C. Kanellakis, P. Ragde, A.A. Shvartsman: Parallel algorithms with processor failures and delays, J. Algorithms, 20, 45–86 (1996)
2. B.S. Chlebus, R. De Prisco, A.A. Shvartsman: Performing Tasks on Restartable Message-Passing Processors, In: Proc. 11th International Workshop on Distributed Algorithms, Saarbrücken, Germany, Lecture Notes in Computer Science 1320, pp. 96–110, Berlin Heidelberg New York: Springer 1997
3. B.S. Chlebus, D.R. Kowalski: Randomization Helps to Perform Tasks on Processors Prone to Failures, In: Proc. 13th International Symp. on Distributed Computing, Bratislava, Slovakia, Lecture Notes in Computer Science, Berlin Heidelberg New York: Springer 1999
4. Communications of the ACM, Special Issue on Group Communication Services, Vol. 39, No. 4, 1996
5. R. De Prisco, A. Mayer, 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
6. C. Dwork, J. Halpern, O. Waarts: Performing Work Efficiently in the Presence of Faults, SIAM J. on Computing, 27(5), 1457–1491, 1998. (Preliminary version appears as "Accomplishing Work in the Presence of Failures" in Proc. of the 11^{th} ACM Symp. on Principles of Distr. Comp., pp. 91–102, 1992)
7. Z. Galil, A. Mayer, M. Yung: Resolving Message Complexity of Byzantine Agreement and Beyond, In: Proc. 36th IEEE Symposium on Foundations of Computer Science, pp. 724–733, 1995
8. V. Hadzilacos, S. Toueg: Fault-Tolerant Broadcasts and Related Problems, In: S. Mullender (ed.) Distributed Systems (2nd edn), Addison-Wesley and ACM Press, 1993
9. P.C. Kanellakis, D. Michailidis, A.A. Shvartsman: Controlling Memory Access Concurrency in Efficient Fault-Tolerant Parallel Algorithms, Nordic Journal of Computing, 2, 146–180, 1995 (Preliminary version in Proc. 7th International Workshop on Distributed Algorithms, pp. 99–114, 1993)
10. P.C. Kanellakis, A.A. Shvartsman: Efficient Parallel Algorithms Can Be Made Robust, Dist. Comput. 5, 201–217, 1992. (Prel. version in Proc. of the 8th ACM Symp. on Principles of Distributed Computing, 1989, pp. 211–222)
11. P.C. Kanellakis, A.A. Shvartsman: Fault-Tolerant Parallel Computation, ISBN 0-7923-9922-6, Dordrecht: Kluwer Academic Publishers 1997
12. Z.M. Kedem, K.V. Palem, P. Spirakis: Efficient Robust Parallel Computations, In: Proc. 22nd ACM Symp. on Theory of Computing, pp. 138–148, 1990.
13. 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
14. C. Martel, A. Park, R. Subramonian: Work-Optimal Asynchronous Algorithms for Shared Memory Parallel Computers, SIAM J. Comput. 21 (1992) 1070–1099. (Prel. version appears as 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)
15. R.D. Schlichting, F.B. Schneider: Fail-stop processors: An approach to designing fault-tolerant computing systems, TOCS 1, 3 (August 1983), 222–238

Bogdan S. Chlebus received a Master of Science degree in mathematics from the Warsaw University (Poland) in 1977, and a Ph.D. in computer science from the Warsaw University in 1982. He is on the faculty of Institute of Informatics at the Warsaw University. His primary research interests are in distributed and parallel computing. Recently he has concentrated on the issues of communication in networks, dependable computing, and efficient use of randomization. Among his publications is a chapter on randomized communication in radio networks, included in the handbook of randomized computing, to be published by Kluwer Academic.

Roberto De Prisco received a *Laurea* degree from the University of Salerno (Italy) in 1991, a Master of Science from the Massachusetts Institute of Technology in 1997, a *Dottorato* degree from the University of Naples (Italy) in 1998 and a Ph.D. from the Massachusetts Institute of Technology in 2000, all of them in Computer Science. Currently he is a member of the Algorithms group at the University of Salerno, Italy, and a member of the Theory of Distributed Systems group at the Laboratory for Computer Science, MIT. His research interests focus on distributed systems and algorithms and data structures.

Alex Allister Shvartsman is on the faculty of Computer Science and Engineering at the University of Connecticut and he is a research affiliate at the MIT Laboratory for Computer Science. His primary research interests are in the principles and practices of dependable distributed and parallel computing. He has published many articles on these subjects and is the author of the monograph *Fault-Tolerant Parallel Computation* (with P.C. Kanellakis, Kluwer Academic, 1997). He has also led the development of several distributed systems in manufacturing automation, resource management and interactive multimedia. He received a B.S. from Stevens Institute of Technology in 1979, a M.S. from Cornell University in 1981, and a Ph.D. from Brown University in 1992, all in Computer Science.